



UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Faculty of Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Software Engineering Group
Warburger Straße 100
33098 Paderborn

Scenario-based Design of Mechatronic Systems

by

Joel Greenyer

jgreen@upb.de

PhD Thesis

in partial fulfilment of the requirements for the degree of
doctor rerum naturalium (Dr. rer. nat.)

supervised by

Prof. Dr. Wilhelm Schäfer

Paderborn, October 11, 2011

Abstract

Mechatronic systems today have to fulfill increasingly complex tasks in diverse and often safety-critical situations. In order to cope with this complexity, the design of the system is typically based on scenarios, in which the engineers describe which sequences of events may, must, or must not happen in certain situations. Scenarios allow humans to conceive complex requirements. However, it may happen that contradictions are introduced among the scenarios, and thus the specification becomes inconsistent. If such inconsistencies are not detected early, this may require costly iterations in the system's development or it may lead to flaws in the system.

In the scope of this thesis, a method was developed for finding inconsistencies in scenario-based specifications of mechatronic systems. Modal Sequence Diagrams (MSDs) were extended so that now real-time requirements and environment assumptions can be formulated. A technique was developed that maps the problem of finding inconsistencies in such MSD specifications to the problem of synthesizing winning strategies in two-player games. This way, an existing, efficient algorithm can be employed for consistency analysis. Moreover, this thesis presents a formal technique for decomposing the synthesis problem.

Furthermore, in order to find inconsistencies in specifications of dynamic systems, this thesis presents concepts for improving the play-out algorithm. This algorithm allows for the simulation of MSD specifications, but it may run into avoidable violations of the specification. By combining the play-out with strategies that could be successfully synthesized from parts of the specification, the simulation produces less avoidable violations, thus giving the engineers more reasons to suspect inconsistencies if violations occur.

Zusammenfassung

Mechatronische Systeme müssen heute immer komplexere Aufgaben in vielseitigen, teils sicherheitskritischen Situationen erfüllen. Um diese Komplexität zu beherrschen, basiert der Entwurf solcher Systeme meist auf Szenarien, in denen Ingenieure beschreiben, welche Folgen von Ereignissen in bestimmten Situationen passieren können, müssen oder nicht passieren dürfen. Durch Szenarien können Menschen komplexe Anforderungen erfassen. Allerdings können Widersprüche zwischen den Szenarien entstehen und die Spezifikation somit inkonsistent werden. Werden Inkonsistenzen nicht früh entdeckt, kann dies teure Iterationen in der Entwicklung erfordern oder zu Fehlern im System führen.

In dieser Dissertation wurde eine Methode entwickelt, um Inkonsistenzen in szenariobasierten Spezifikationen mechatronischer Systeme zu finden. Modal Sequence Diagrams (MSDs) wurden erweitert, sodass nun auch Echtzeiteigenschaften und Umweltannahmen beschrieben werden können. Eine Technik wurde entwickelt, welche das Finden von Inkonsistenzen auf die Synthese von Gewinnstrategien in Zweispielerproblemen abbildet, sodass ein existierender, effizienter Algorithmus für die Konsistenzanalyse verwendet werden kann. Zudem wurde eine Technik für die Dekomposition des Syntheseproblems entwickelt.

Um Inkonsistenzen in Spezifikationen von dynamischen Systemen zu finden, werden zudem Konzepte für die Verbesserung des play-out-Algorithmus präsentiert. Der existierende Algorithmus ermöglicht die Simulation von MSD Spezifikationen, bei der es jedoch zu vermeidbaren Verletzungen der Spezifikation kommen kann. Durch eine neuartige Kombination des play-out-Algorithmus mit erfolgreich für Teile der Spezifikation synthetisierten Strategien produziert dieser weniger vermeidbare Verletzungen. So hat der Ingenieur mehr Grund eine Inkonsistenz zu vermuten, wenn Verletzungen auftreten.

Acknowledgment

This work would not have been possible without the great working atmosphere at the software engineering group in Paderborn. Chiefly responsible for that is Wilhelm Schäfer, who I especially thank for his confidence in me, his good advice, and the space he provided that allowed me to develop ideas and to drive my research in a self-dependent way. Second, although not longer part of the software engineering group in Paderborn, I thank Ekkart Kindler, who has taught me many things and who has sparked my fascination for software engineering.

I also thank all my current and former colleagues at the software engineering group for the helpful discussions and the good humor (especially during the coffee breaks and any time “after 4pm”), especially Oliver Sudmann, Jan Rieke, Matthias Meyer, Dietrich Travkin, Martin Hirsch, Stefan Henkler, Matthias Tichy, Claudia Priesterjahn, Markus von Detten, Tobias Eckhard, Renate Löffler, Christian Heinzemann, Jan Meyer, Jörg Holtmann, Björn Axenath, Patrick Könemann, David Schmelter, Steffen Becker, and Christian Brenner. Especially supportive during the last years were Jutta Haupt (not only the administrative support, great were also the occasional cooky supply and the good laughs), Jürgen Maniera (a.k.a. “Sammy”), Astrid Canisius, and Eckhard Steffen. Also, Christian Brenner and Yvonne Makopa helped a lot with proofreading.

Furthermore, I thank Gregor Engels and the members of his group for the helpful discussions during the reading classes. I thank Heike Wehrheim for her good advice. Very interesting and inspiring was the interdisciplinary work with Prof. Gausemeier and his group. I especially thank Sascha Kahl, Roman Dumitrescu, Sebastian Pook, and Jörg Donoth for the productive and fun cooperation. Also, I thank Jens Geisler and Christian Henke for the detailed insights on the RailCab project. Very interesting and helpful was furthermore the extensive discussion on MSDs with Shahar Maoz and the correspondence on UPPAAL TIGA with Shuhao Li. It was especially fun working with Sascha Burdick, Nils Diekmann, Jens Frieben, Markus Fockel, Mathias Höckelmann, Liliya Klassen, Daniel Loechelt, and Daniel Simon in the SCENARIOTOOLS project group.

I thank my family for their constant care and support: my parents Ruth Greenyer-Boekstegers and Barnabas Greenyer, my sister Manon Greenyer, and Nikolaus Boekstegers. For their endless care, I especially thank my grandparents Elfriede and Benno Herzog. Finally, I thank Judith Mayer, my love, partner, and friend, for her encouragement, patience, and for being there for me all the time.

Short contents

1	Introduction	· 1
2	Problem Analysis	· 11
3	Foundations	· 23
4	Synthesis	· 57
5	Symbiosis of Simulation and Synthesis	· 117
6	Triple Graph Grammar Extensions	· 143
7	Realization and Evaluation	· 159
8	Related Work	· 177
9	Conclusion and Future Research	· 185
A	Meta-Models and Profiles	· 191
B	MSD-to-TGA TGG Transformation	· 199
C	Examples	· 231
	Bibliography	· 277
	List of Figures	· 289
	Index	· 297

Contents

1	Introduction	1
1.1	The problem	2
1.2	The objective	6
1.3	The approach	6
1.4	The contribution	8
1.5	The structure of this thesis	9
2	Problem Analysis	11
2.1	Characteristics of mechatronic systems	11
2.1.1	The architecture of advanced mechatronic systems	11
2.2	The development of mechatronic systems	13
2.2.1	The interdisciplinary design language elaborated in the CRC 614	13
2.2.2	Example: specifying two use cases in the RailCab system	14
2.3	Problem description	15
2.4	Existing scenario-based design techniques	19
3	Foundations	23
3.1	Modal Sequence Diagrams	23
3.1.1	MSDs and the object system	23
3.1.2	Events, messages, and runs	25
3.1.3	Event unification	25
3.1.4	Existential and universal MSDs, hot and cold messages	26
3.1.5	Active MSDs, the cut, and hot and cold violations	26
3.1.6	The iterative interpretation of MSDs	27
3.1.7	Satisfying an MSD specification	30
3.1.8	The play-out algorithm	30
3.1.9	Consistency, consistent executability, and realizability	32
3.1.10	Parameterized messages	33
3.1.11	Object properties and side-effects	34
3.1.12	Assignments and conditions	35
3.1.13	Visible and Hidden events	36
3.1.14	Timed MSD specifications	37
3.1.15	Symbolic lifelines	39
3.1.16	Forbidden messages	45

3.2	Timed Game Automata and Uppaal Tiga	46
3.2.1	Timed Automata in UPPAAL	47
3.2.2	Timed Game Automata in UPPAAL TIGA	49
3.2.3	On-the-fly synthesis of game strategies	49
3.3	Triple Graph Grammars	50
3.3.1	Why model transformation with TGGs?	51
3.3.2	TGG structure and semantics	51
3.3.3	Forward transformation	54
3.3.4	Further extensions of TGGs	56
4	Synthesis	57
4.1	Overview	58
4.2	The MSD specification scheme	59
4.2.1	Collaboration and class diagram	59
4.2.2	Requirement MSDs and assumption MSDs	60
4.3	Mapping untimed MSD specifications	62
4.3.1	The environment and system automata for untimed MSD specifications	62
4.3.2	Mapping the MSDs to TGA	67
4.3.3	Encoding assignments and conditions	73
4.3.4	Forbidden messages	76
4.3.5	Assumption MSDs	77
4.4	The winning condition	79
4.4.1	Checking consistent executability with UPPAAL TIGA	79
4.4.2	An alternative winning condition	81
4.5	Mapping timed MSD specifications	82
4.5.1	The environment and system automata for timed specifications	82
4.5.2	Encoding clock resets and time conditions	86
4.5.3	Extensions to the winning condition for timed spec- ifications	96
4.6	Compositional synthesis	97
4.6.1	Compositional reasoning	97
4.6.2	The compositional synthesis technique	99
4.6.3	Example: the compositional synthesis of the produc- tion cell specification	103
4.6.4	The compositional synthesis technique is sound	108
4.7	Different kinds of consistency	113
4.7.1	Disallowing to delay steps in the timed setting	113
4.7.2	Consistency vs. consistent executability	114
4.8	Summary and Outlook	115
4.8.1	Inconsistent environment assumptions	115
4.8.2	Partial observability	115
5	Symbiosis of Simulation and Synthesis	117
5.1	Overview	117
5.2	Guiding by controllers from single use cases	120

5.2.1	Use case specification example	120
5.2.2	Play-out with synthesized controllers	123
5.3	Guiding by controllers from composed use cases	128
5.3.1	Composed use case example	128
5.3.2	Synthesizing controllers from composed use cases	131
5.3.3	Guiding the play-out of composed use case occurrences	132
5.3.4	Tracking composed use case occurrences	133
5.3.5	Systematically tracking composed use case occurrences	137
5.3.6	Overly restrictive context expressions	140
5.4	Summary and Outlook	140
6	Triple Graph Grammar Extensions	143
6.1	Overview of the TGG extensions	143
6.2	Generalization of TGG rules	144
6.2.1	Why a generalization concept for transformation rules?	145
6.2.2	The TGG rule generalization concept by Klar et al.	146
6.2.3	Improvements to the existing TGG rule generalization concept	146
6.3	OCL attribute value constraints	150
6.4	UML stereotype constraints	152
6.5	Reusable nodes	152
6.5.1	Reusable nodes in the example	153
6.5.2	The operational semantics of reusable nodes in the target domain during a forward transformation	154
6.6	Summary	157
6.7	Outlook	157
7	Realization and Evaluation	159
7.1	The TGG Interpreter	159
7.2	MSD-to-TGA mapping and synthesis	161
7.3	ScenarioTools	162
7.3.1	Modeling	162
7.3.2	Simulation	165
7.3.3	Physics-engine and visualization	169
7.4	Evaluation	169
7.4.1	Practicality of the MSD formalism	170
7.4.2	Synthesis	173
7.4.3	Simulation	174
7.4.4	TGG-based model transformation	175
8	Related Work	177
8.1	Advanced play-out techniques	177
8.2	Related synthesis approaches	178
8.2.1	Synthesis of global controllers	179
8.2.2	Synthesis of distributed controllers	182
9	Conclusion and Future Research	185

9.1	Summary	185
9.2	Future research	187
A	Meta-Models and Profiles	191
A.1	The meta-model for UPPAAL TIGA	191
A.2	MSD specifications in UML	193
B	MSD-to-TGA TGG Transformation	199
B.1	TGG rule overview	199
B.2	OCL attribute definitions	201
B.3	TGG rules	204
C	Examples	231
C.1	Simulating an example RailCab specification	232
C.1.1	Example specification overview	232
C.1.2	Use case DriveOntoTrackSection	234
C.1.3	Use case DriveOntoBranchingSwitch	238
C.1.4	Use case EnergyManagement	242
C.1.5	The simulation model	247
C.2	The symbiosis of synthesis and simulation – the use case “Warn RailCabs On Track”	250
C.2.1	Description of the use case “Warn RailCabs On Track”	250
C.2.2	The specification of the use case	250
C.2.3	Avoidable violating runs	252
C.2.4	The controller synthesized from the use case specifi- cation	252
C.3	Synthesis example – the production cell	257
C.3.1	Description of the production cell example	257
C.3.2	The MSD specification of the production cell	258
C.3.3	Compositional synthesis of the production cell spec- ification	263
C.4	Synthesis performance measurement	271
C.4.1	How the measurements were taken	271
C.4.2	Untimed specifications with exponentially growing state space	271
C.4.3	Timed specification with exponentially growing state space	274
	Bibliography	277
	List of Figures	289
	Index	297

Introduction

From household machines to medical devices and transportation systems, we find mechatronic systems everywhere around us today. Mechatronic systems also drive our industry in areas such as manufacturing, logistics, and agriculture. To serve customers' needs and to make systems more cost- and energy efficient, these systems have to fulfill increasingly advanced functions and must often be able to autonomously control complex physical processes. In some cities or at airports, for example, we already find driverless transportation systems in operation. Other examples are the highly automated logistics systems in harbors and distribution centers.

The new requirements of today's systems are often met by building better hardware, for example better sensors or more efficient drives. But, the advanced features of these systems are essentially enabled by software that controls the complex processes in the systems. Software also allows for the communication of subsystems and modules that were separate before, and therefore more and more advanced functions are realized by the autonomous collaboration of mechatronic modules and subsystems. Research today shows that it is even possible to build systems that react robustly to hardware failures or systems that monitor their performance and adapt their behavior autonomously to changing environmental conditions. The former systems are called *self-healing* systems, the latter are called *self-optimizing* systems.

Self-optimizing mechatronic systems and techniques for successfully developing such systems are the focus of the Collaborative Research Center 614 "Self-optimizing Concepts and Structures in Mechanical Engineering" (CRC 614), a large-scale interdisciplinary research program at the University of Paderborn. This thesis is embedded in this research program.

There are two aspects that make the development of mechatronic systems especially challenging. First, these systems are often used in safety-critical areas. To make the systems safe, significant development time and costs has to be spent on testing today. Second, the mechanical, electrical, and software aspects of mechatronic systems are highly interrelated and can no longer be developed

separately. The growing number of requirements, the interconnectedness of the mechatronic subsystems and modules, and the interdisciplinary nature of these systems generate an enormous complexity that engineers must master during the development of these systems. To handle the increasing complexity, systematic development processes on the one hand, and *model-based development techniques* on the other hand are developed within the CRC 614.

Model-based development techniques are more commonly applied in practice today, which means that engineers use problem-specific models to describe the relevant concerns during the development of a system. In the automotive domain, for example, car manufacturers and suppliers are beginning to adopt the AUTOSAR standard for describing the software architecture of cars in a standardized way. Now parts of the software component's code can be generated automatically, which avoids mistakes due to manual programming and prevents many problems of integrating software components later on. Also, the code for the continuous control functions is often generated from models today, for example from MATLAB-Simulink block diagrams.

But, although manufacturers recognize the benefits of model-based development techniques, not all aspects of a system are described by adequate models in practice today. This is especially true for the communication behavior of mechatronic modules and subsystems. Moreover, model-based development techniques are often applied only at later phases of the development. During the early design, the specification of the system and its software is typically captured in the form of informal diagrams and text. This implies that an automated analysis of the design is only possible at later phases of the development. Testing often only takes place after the components and test cases have been implemented. However, when errors are detected during testing, they sometimes reveal that the specification was *inconsistent* in the first place and it was in fact never possible to come up with an implementation that could satisfy all the requirements in all operating conditions. If such inconsistencies are not detected early, resolving them often requires costly iterations.

Therefore, to reduce the development costs, new development techniques are required that allow the engineers to specify the system and software requirements in an understandable, but precise way. Especially, the engineer must be supported by automated analysis techniques already in the early design phases in order to find inconsistencies in time.

1.1 The problem

For the design of mechatronic systems, the VDI 2206 "Design Methodology for Mechatronic Systems" [VDI04] was developed, which addresses the particular challenges in the interdisciplinary development of mechatronic systems. It proposes a development process, which is based on the *V-model*. The V-model divides the system development into three main development phases that can be displayed in a V-shaped form. A V-model adapted from the VDI 2206 is illustrated in Fig. 1.1. During the first development phase, called the *conceptual design*, experts from all disciplines jointly develop a first interdisciplinary

conceptual model of the system, called the *principle solution*. The principle solution is the basis for the subsequent discipline-specific *implementation* phase. In this phase, the disciplines develop the discipline-specific parts of the system, using their discipline-specific methods and tools. In the third phase, the *system integration*, the discipline-specific results are integrated to form the final product, while the components on the different hierarchy levels are validated and verified against the requirements that were specified during the conceptual design.

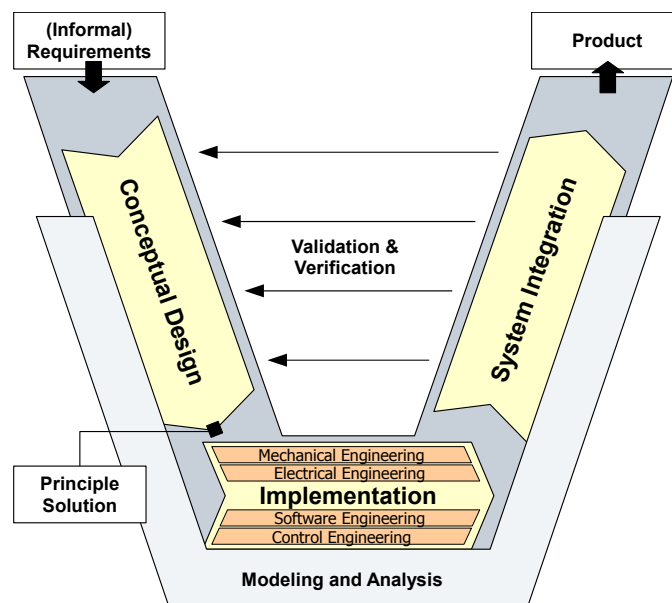


Figure 1.1: A V-model describing the development of a mechatronic system, on the basis of the VDI 2206 “Design Methodology for Mechatronic Systems”

In the early conceptual design of a system, the engineers and stakeholders have to develop a common understanding of the system to be developed. This is typically done by gathering the different *use cases* of the system. A use case describes the functionality of a system by sequences of interaction events that take place among system and environment elements. These sequences may be sequences that are required or possible to occur, or exceptional and erroneous sequences [Jac92, RJB05, Poh07, vL09]. Such sequences of events are commonly called *scenarios*.

For humans, use cases and scenarios are a natural way to conceive and communicate what the system can, must, or must not do. They allow engineers to concentrate on specifying one function of the system at a time, instead of having to regard the whole functionality of the system at once. Therefore, use cases and scenarios are an intuitive basis for capturing the complex requirements of a system. A design that is based on use cases and scenarios is also called a *scenario-based* design.

However, there are two problems that occur during the scenario-based design. First, the scenarios are typically described informally today. This im-

plies that the requirements posed by the scenarios may be ambiguous and may therefore later be interpreted differently than originally intended. An interdisciplinary specification language was developed within the CRC 614, that allows engineers to capture the descriptions of a scenario by semi-formal structure and behavior diagrams [Fra06]. Still, it turns out that the means for capturing the behavior described in scenarios are inadequate. The same is true also for other languages that may be used in the early system design, such as UML [UML09] or SysML [Sys08].

The second problem is that, even when capturing the scenarios formally, there may easily be contradictions in how the system is intended to behave. Especially if the different use cases are formulated by different engineers or stakeholders, there may easily be two scenarios, describing the same situation, where one scenario requires the system to act in a way that is forbidden by the other scenario. If that is the case, a specification is said to be *inconsistent*.

Let us consider an example in the following. The examples considered here are taken from the context of the project “Neue Bahntechnik Paderborn/RailCab”¹, *RailCab* in short. RailCab is an innovative concept for the future rail-bound traffic. The vision is that, in the future, trains will no longer run on a schedule, but small, autonomous vehicles, called *RailCabs*, transport passengers and goods on demand. To reduce the energy loss due to wind resistance, the RailCabs may form convoys when traveling in the same direction. Figure 1.2 illustrates the vision of the RailCab project. The images on the right show the RailCab test track that is built at a scale of 1:2,5 at the university campus in Paderborn.

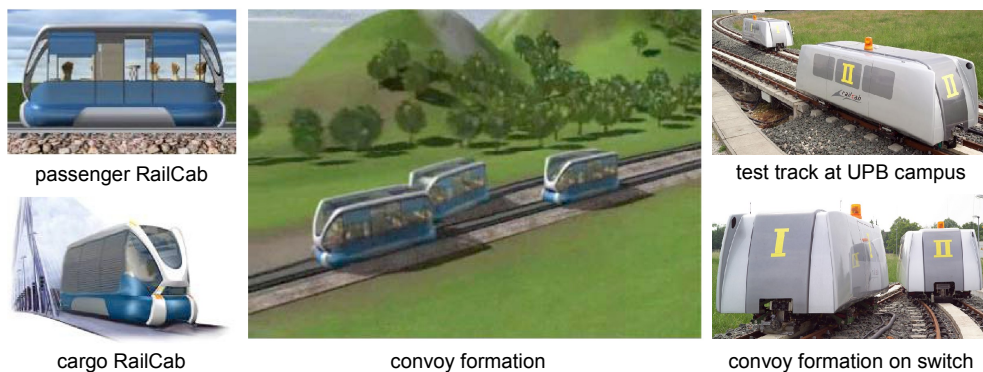


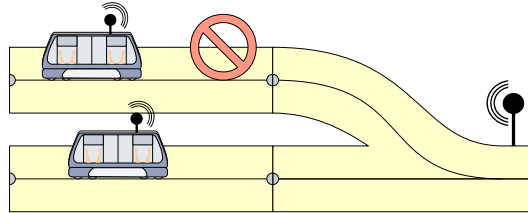
Figure 1.2: The RailCab system: On-demand transportation of passengers and goods by autonomous rail vehicles (taken from [ADG⁺09])

During the early design of such a system, engineers and other stakeholders collect a number of use cases. For example, there may be different use cases for a scenario where two RailCabs arrive at a merging switch, see Fig. 1.3. One engineer may specify the standard case (i): in order to avoid collisions, a RailCab must not enter a merging switch if another RailCab (that arrived earlier) is already granted the permission to enter the switch. Another stakeholder

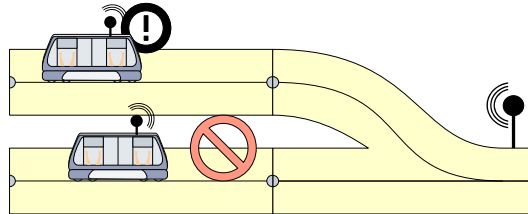
¹“Neue Bahntechnik Paderborn”, <http://www-nbp.upb.de>

(ii) wishes that express RailCabs with first-class passengers shall be given the right of way before regular RailCabs. Third (iii), an engineer has in mind that switches are only short sections of track and that the performance of the linear drive to break on switches may be very limited. Therefore, the case has to be considered that there has been a hazard reported on the subsequent track section. Then none of the approaching RailCabs are allowed to enter the switch.

(i) The later RailCab must wait:



(ii) Certain kinds of RailCabs have the right of way:



(iii) Entering is denied when a hazard occurred on the subsequent track section:

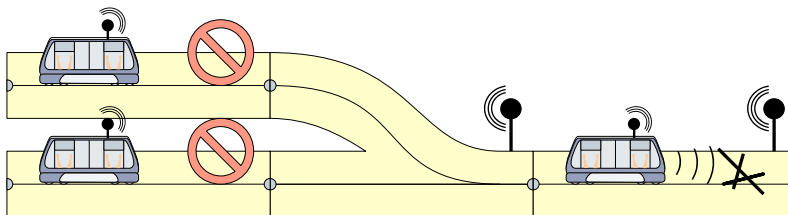


Figure 1.3: Three use cases of RailCabs entering a merging switch

Between these use cases, there may be for example the following contradiction: The requirements for use case (i) may be formulated in an overly restrictive way. It may state that the RailCab arriving first must be permitted to enter the switch. However, both use cases (ii) and (iii) describe a situation where this must not happen. This is of course just a simple example, but inconsistencies may be much more subtle. Sometimes contradictions only occur in rare cases. Nevertheless, violating the requirements even in rare cases may have devastating consequences. The more use cases describe specific concerns in the same situations, the more likely it is for inconsistencies to occur. To the use cases above, for example, further ones will be added that describe how RailCabs shall be able to form convoys or merge convoys at switches (see picture in the middle of Fig. 1.2).

The dependencies among the use cases are often very complex and it is very difficult to detect all contradictions by manual revision. But not detecting these contradictions right away may lead to costly iterations later in the development.

If these contradictions even remain undetected, it may lead to critical flaws in the final product. Especially in the development of mechatronic systems, it is important that the principle solution forms a consistent basis for the implementation phase, because detecting and resolving inconsistencies spanning the different interdisciplinary implementations is a great challenge today [GSG⁺09].

1.2 The objective

In order to address the above problems, the objective of this thesis is to develop a technique that supports the engineer in the intuitive, but precise specification of scenario-based requirements during the early conceptual design. Furthermore, the engineer shall be supported by automated techniques in detecting inconsistencies among the requirements as early as possible.

The focus here lies on the requirements on the *discrete behavior* of a system, because many central features of advanced mechatronic systems are based on the discrete interaction of system components. Moreover, interdisciplinary behavioral concerns are often described on an abstract, discrete level during the early design of a system.

Furthermore, *time requirements* shall be regarded. Time is an important aspect in the design of mechatronic systems, because the software in such systems often has to control physical processes where it is crucial to consider delays between certain control operations.

1.3 The approach

This thesis presents a technique for the consistent scenario-based design of mechatronic systems. The technique is based on *Modal Sequence Diagrams* (MSDs) [HM08], a formal interpretation of UML Sequence Diagrams [UML09] for describing sequences of events that may, must, or must not occur during the interaction of the system components and their environment. The concepts of MSDs are based on *Live Sequence Charts* (LSCs), developed by Damm and Harel [DH98, DH01]. In order to detect inconsistencies in MSD specifications, this thesis proposes a combination of *simulation* and *formal synthesis*, as described in the following.

When an engineer specifies the scenarios by a certain kind of MSDs, these scenarios can be executed by an algorithm called the *play-out* algorithm [HM03]. The play-out algorithm allows engineers to simulate the behavior that emerges from the interplay of multiple scenarios. This is a powerful technique to acquire an insight into the relationships among the scenarios and to detect inconsistencies in the specification. But by using simulation alone, many inconsistencies may remain undetected. There are two reasons for that. First, even systematic testing will typically not cover all possible situations that may occur during the operation of the system. Second, the simulation algorithm may at times report violations of the requirements even if the requirements are consistent. This is because, during the early design of the system, there is typically a lot of non-determinism remaining in the specification; we then say that the specification is

under-specified. In under-specified specifications, the play-out algorithm has to make many non-deterministic choices. This means that if a certain sequence of steps leads to a violation of the requirements, there may be another sequence of steps that the play-out algorithm could have chosen to avoid the violation. The play-out algorithm, however, cannot “look ahead” to avoid the avoidable violations. Therefore, if violations occur during the simulation, the engineer cannot be sure whether the requirements are really inconsistent or only the play-out algorithm fails to execute them consistently.

The *formal synthesis*, by contrast, is a technique that allows the engineer to find out for sure whether the requirements described by a set of MSDs are consistent or not. A synthesis algorithm answers this question by trying to find a controller that satisfies the requirements under consideration of every possible sequence of events in the system’s environment. If the synthesis algorithm can find such a controller, it means that the regarded set of requirements is consistent; if it fails to find an implementation, the requirements are inconsistent.

The disadvantage of formal synthesis is that it is an inherently complex task. Therefore, it can only be applied to specifications of limited size. Furthermore, the synthesis is not suited for finding controllers for *dynamic* systems. Dynamic systems are systems in which the communication relationships between the components may change over time or components may be added to or removed from the system. This leads to an additional explosion of the number of states that would have to be considered, which makes the synthesis more difficult or even impossible if the number of possible system configurations is infinite. Therefore, we have to rely on the simulation via the play-out algorithm for finding inconsistencies in larger specifications or dynamic systems.

In order to more effectively find inconsistencies by the simulation via the play-out algorithm, the approach in this thesis is to *combine the synthesis and simulation* in a beneficial way: First, the synthesis is employed to detect inconsistencies in single use cases or combinations of use cases that refer to *static situations* during the runtime of the system. Static situations are such situations where the communication relationships between the regarded components do not change. Consider for example the use cases shown in Fig. 1.3. These use cases refer to a static situation in the RailCab system where one or two RailCabs approach a merging switch. For the duration of the use cases, there is a fixed set of objects with fixed communication relationships.

If the synthesis detects inconsistencies, these inconsistencies must be resolved by the engineers. If the synthesis is successful, it yields a controller that shows how the system can satisfy the regarded part of the specification. Unfortunately, even if controllers can be successfully synthesized for all use cases in a specification, this does not imply that the specification is consistent. That is because use cases may still “overlap” in different ways. Therefore, the simulation via play-out is still useful and necessary to find inconsistencies in this case. In order to avoid more avoidable violations during play-out, the approach is to improve the play-out by also interpreting the controllers that could be successfully synthesized from the single use case specifications. This reduces the amount of avoidable violations during simulation and thus gives the engineers more reason to suspect an actual inconsistency if a hot violation occurs.

1.4 The contribution

The contribution of this thesis is as follows:

A novel synthesis technique. This thesis presents a novel technique for synthesizing controllers from timed and untimed MSD specifications. The synthesis is realized by mapping an MSD specification to the input for UPPAAL TIGA, a tool for finding winning strategies in untimed and timed two-player games [CDF⁺05, BCD⁺06, BCD⁺07a]. A similar approach was presented by Larsen et al. [LLNP09, LLNP10], which, however, has a number of limitations. For example, their approach does not consider many sequences of environment events, which may nevertheless occur in reality. Thus, critical sequences of environment events may be overlooked during the synthesis. The particular novelty of the technique presented here is that it considers *environment assumptions* that can be specified by an engineer. Being able to consider specific environment assumptions is crucial in practical applications, because often a system will only be able to satisfy its requirements if it can be assumed that not arbitrarily “bad” things can happen in its environment. Especially in mechatronic systems, many system functions are only realizable under the assumption that mechanical parts and the laws of physics restrict the possible sequences of environment events. The environment assumptions can be formalized in the form of *assumption MSDs*, by which the engineer can precisely specify that after a particular sequence of events, certain events will or will not occur in the environment. This is especially convenient during the early design, if only incomplete information on the environment is available.

A compositional synthesis technique. Based on the concept of assumption MSDs, this thesis furthermore presents a novel technique that, in certain cases, allows engineers to *decompose* an MSD synthesis problem into two or more smaller synthesis problems that in sum can typically be solved much faster than the original synthesis problem. This technique is based on the *assume-guarantee* paradigm [MC81, Jon83, Pnu85, GL94, CGP99] and consists of a number of manual steps. These steps describe how to split a given MSD specification into two parts and how to add additional MSDs as assumptions and requirements to these parts, so that it is valid to imply the consistency of the original specification from the consistency of the parts. A sketch of a proof is presented that shows the soundness of the technique. Furthermore, the technique is validated by the MSD specification of an industrial production robot. This kind of compositional synthesis of LSC/MSD specifications has not been studied previously. Kugler and Segall also propose a technique for the “compositional synthesis” of LSC specifications [KS09]; their approach, however, differs significantly to the one presented in this thesis.

Symbiosis of play-out and synthesis. This thesis presents concepts for improving the simulation of an MSD specification via the play-out algorithm with synthesized controllers for single use cases or combinations of use cases in the specification. The concept is validated by an example. Kugler et al. mention that their approach supports the execution of a controllers

synthesized for an LSC specification [KPP09]. However, executing a synthesized controller in combination with the play-out other LSCs/MSDs has not been regarded thus far.

Novel extensions to TGG-based model transformations. The mapping from an MSD specification to the input for UPPAAL TIGA is very complex, which makes it difficult to formally specify this mapping and to develop an automatic translator. In this thesis, the mapping is formalized and implemented by a *Triple Graph Grammar* (TGG) [Sch94]. TGGs are a declarative, rule-based formalism for specifying the relationship between models, and are therefore an intuitive formalism for specifying the mapping. Furthermore, a TGG can be interpreted for automatically transforming a given input model into a target model. In order to adequately describe the mapping, a number of extensions to the TGG formalism were elaborated in the scope of this thesis. First, the *Object Constraint Language* (OCL) [OCL10] was integrated in TGGs. Second, a rule generalization (inheritance) concept for TGG rules was elaborated, which improves the generalization concept previously proposed by Klar et al. [KKS07]. As a result, the mapping is specified in a concise way and the same mapping specification can furthermore be automatically executed.

The central concepts presented in this thesis were prototypically implemented and validated by examples. The extensions to the TGG formalism are implemented in the TGG INTERPRETER. An MSD editor, the transformation which maps an MSD specification to the input for UPPAAL TIGA, and an improved play-out algorithm that can be guided by synthesized controllers are implemented in the SCENARIOTOOLS tool suite. Also, a 3D-visualization for the simulation of a system of RailCabs was integrated with SCENARIOTOOLS.

1.5 The structure of this thesis

This thesis is structured as follows. Chapter 2 presents a more detailed problem analysis. The foundations are introduced in Chap. 3. Chapter 4 describes the synthesis technique. Chapter 5 presents how the play-out of an MSD specification can be guided by controllers synthesized from single use cases in the specification. It furthermore explains how situations of overlapping use cases can be specified as *composed use cases*, and how the play-out can be guided by controllers synthesized from composed use cases. The extensions to the TGG-based model transformation concepts are described in Chap. 6. Chapter 7 shows how the concepts developed in this thesis were implemented. Chapter 8 discusses the related work, and Chap. 9 concludes this thesis and presents new research perspectives that emerge from this work.

Further examples and technical details can be found in the appendices. Appendix A describes a UML profile for MSD specification as well as a meta-model for UPPAAL TIGA. Appendix B shows the TGG for mapping an MSD specification to the input for UPPAAL TIGA. Finally, Appendix C shows examples of MSD specifications that were used to validate the synthesis and simulation concepts and to benchmark the synthesis.

Problem Analysis

This chapter introduces the problem addressed in this thesis in more detail. First, Sect. 2.1 introduces the characteristics of mechatronic systems. Next, the methodology for the conceptual design of advanced mechatronic systems as it is currently proposed by the CRC 614 is explained in Sect. 2.2. Section 2.3 then illustrates the current problems with the scenario-based design according to this methodology by an example. Last, Sect. 2.4 briefly overviews existing techniques that have been developed to address the problem.

2.1 Characteristics of mechatronic systems

Mechatronic systems are a class of systems that combine mechanical, electronic and software components and therefore require the collaboration of different engineering fields or *disciplines* for their development [VDI04, Ise05, Ise07, Ise09]. These disciplines are, on the one hand, the classical engineering disciplines mechanical engineering and electrical engineering and, on the other hand, control engineering and software engineering. Control engineering is usually responsible for developing the software components which can be best described by continuous functions. Software engineering is rather concerned with software that is best described by discrete models, such as the interaction behavior between different components and subsystems in a mechatronic system.

2.1.1 The architecture of advanced mechatronic systems

An advanced mechatronic system like the RailCab (as introduced in Chap. 1) typically consists of a hierarchy of interacting mechatronic modules and subsystems. In the CRC 614, the hierarchy levels of mechatronic systems are classified according to Lückel et al. [LHLH01]. In this classification, a mechatronic system consists of several *mechatronic function modules (MFM)*. Each MFM has a mechanical base structure, it has sensors, actuators, and one or several microprocessors to run the software that controls the behavior of the module. Examples

for such a module are the *linear drive module* of a RailCab or its *active suspension and tilting module*. Several MFMs may be combined to an *autonomous mechatronic system (AMS)*. An AMS, for example a single RailCab, has its own energy supply and may interact with its environment autonomously, i.e., without the constant interaction with a human. On the AMS level, the modules are typically supervised and coordinated hierarchically. The modules are for example monitored for failures and measures are taken when failures occur in order to guarantee a safe operation of the system. A number of AMS that jointly fulfill a certain task is called a *networked mechatronic system (NMS)*. On the NMS level, the single members must again be monitored for failures. An example of an NMS is a track system with RailCabs, track section controls, stations, crossings, etc. that all have to interact in order to achieve the desired functionality, which is, in this case, to transport passengers and goods.

On all these levels, the modules and subsystems communicate extensively. Figure 2.1 for example illustrates different situations in which RailCabs interact autonomously with each other and further components of the RailCab track system. The different interactions are labeled by dashed ellipses. For example, a RailCab may detect another RailCab driving ahead and decide to form a convoy with it (*Form convoy*). RailCabs may also break up the convoy again, for example when one RailCab takes another direction than the other members of the convoy (*Leave convoy*). The track section controls are illustrated as antennae attached to a track section. When a RailCab drives from one track section onto another, it is necessary that it requests the permission to enter the next track section (*Drive onto track section*). For example, a hazard may have been detected by another RailCab on the next track section (*Hazard occurred*) such that, for safety reasons, the RailCab may not enter the track section (*Enter denied when hazard on next track section*). Branching and merging switches are also controlled by special kinds of track section controls, called *merging switch control* and *branching switch control*. A merging switch control must for example ensure that two RailCabs, unless they are coordinating to form a convoy, do not enter the switch at the same time (*Drive onto merging switch*). When a RailCab approaches a branching switch, the RailCab has to decide which route to take (*Drive onto branching switch*).

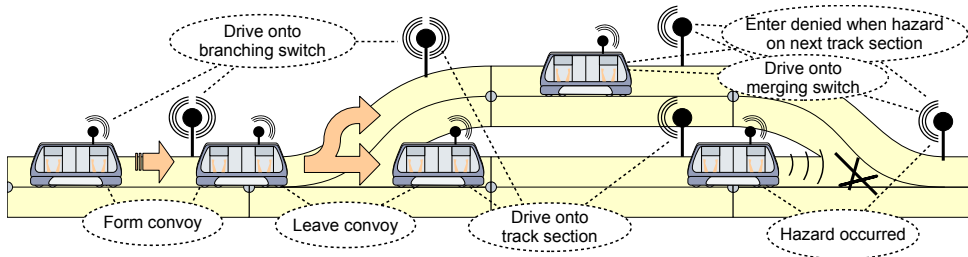


Figure 2.1: Different interactions on the NMS level of the RailCabs system.

2.2 The development of mechatronic systems

Within the CRC 614, a development methodology for advanced mechatronic systems was developed. This methodology first proposes a detailed *development process* [GFDK08a, GFDK08b, GZD⁺08, GFDK09, ADG⁺09]. As already explained in Chap. 1 (see Fig. 1.1, p. 3), the macro-structure of the development process is based on the V-model of the VDI 2206 [VDI04]. In the first phase, the *conceptual design*, experts from all disciplines collaborate to create a first design of the system, called the *principle solution*. For describing this principle solution, the development methodology of the CRC 614 furthermore proposes an interdisciplinary *design language* [Fra06, GFDK08a, GFDK08b, GFDK09], which is similar to UML [UML09] and SysML [Sys08, FMS09], but originates from specific design methods in mechanical engineering that were developed by Kallmeyer [Kal98, GEK01].

2.2.1 The interdisciplinary design language elaborated in the CRC 614

The interdisciplinary design language of the CRC 614 offers engineers a number of diagrams by which many interrelated aspects of a mechatronic system can be described. Figure 2.2 shows an overview of these diagrams. The development process of the CRC 614 proposes that, first, after clarifying the design task and analyzing the environment of the system to be, the use cases of the system are described. For this purpose, the interdisciplinary design language proposes a template for describing use cases by informal text and sketches. Within the CRC 614, a use case is also called *application scenario*, but since *use case* is the established term in computer science, this term is used in this thesis instead.

Based on the use cases, the structure of the system and its environment as well as the behavior of the system is specified in more detail. The structure of the system is captured by *active structure* diagrams. Such a diagram describes the elements of the system and their properties and relationships. Particular kinds of relationships are the flow of information, material and energy between the elements. The *environment* diagram is virtually the same kind of diagram, but it describes the elements in the environment of the system. Here, in particular, the forces and events influencing the system are captured.

The language furthermore specifies behavior diagrams. A special kind of state machine diagrams and a special kind of activity diagrams are proposed to describe the operating states of a system and processes in the system. The behavior diagrams are however not limited to these two [GFDK09], and further behavior diagrams may be added to this language in the future.

The language proposes further kinds of diagrams. The *function hierarchy* can be used to organize the functions of the system in a hierarchical manner and the *requirements* can be used to capture relevant properties of system elements in a structured list. The *shape* diagram can be used to describe a first draft of the three-dimensional structure of the system. The *system of objectives* is a diagram that captures the system's optimization objectives and their relationships, an aspect that is especially important in self-optimizing systems.

The active structure diagrams created for these use cases are very similar. They all specify the logical relationships between the RailCab, the RailCab's current track section control, and the next track section control resp. the control of the merging switch. There are information flow connectors indicating the information exchanged between the environment and system elements.

The activity diagram for the use case **Drive onto track section** shows how the RailCab, after receiving the environment event **approaching end of track section** sends a request to enter the next track section to the next track section control. After that, the next track section control replies, either allowing the RailCab to enter the next track section or not. Then the events occur of the RailCab passing the point of the last safe break, the point of no return, until it eventually enters the next track section. The system elements involved in the activities resp. events are denoted at the top of the events and actions. (This is of course a simplified use case. In practice, we would for example also describe what the RailCab should do if it is not permitted to enter the next track section.)

The activity diagram of the use case **Drive onto merging switch** starts with the reply of the merging switch control to the RailCab. Here a time attribute is added, setting a clock variable *c* to zero. After the reply of the merging switch control, the RailCab passes the point of the last safe break, the point of no return, and finally enters the next track section. The time attribute of the latter event states that it must occur less than 5 seconds after the reply of the merging switch control was sent. (Concepts for representing time in the behavior diagrams of the interdisciplinary conceptual design language have not been precisely defined. The use of clock variables here is an ad hoc extension of the activity diagram to represent time.)

Note that the environment events in these use cases, for example **approaching end of track section**, are not concrete signals from sensors of the RailCab. That is because at this stage of the conceptual design, it is usually not determined by which sensors and actuators the system will interact with its environment. It is a later design choice whether the RailCab determines particular points on the track section for example via fixed markers on the track, or whether it will calculate these points based on its velocity, the traveled distance on the track section, its mass, etc.

2.3 Problem description

The diagrams in Fig. 2.3 and 2.4 intuitively represent the informally described behavior in the use cases, but their meaning is not clear. For example, the diagrams do not precisely answer the following questions:

What is the relationship between the behavior diagrams? The first problem with the above diagrams is that the relationship between the behavior diagrams is not clear. For example, if a RailCab approaches a merging switch, should the system also behave as described in the use case **Drive onto track section**? And if the behavior described in the use case **Drive onto track section** applies, is the action **entering of merging**

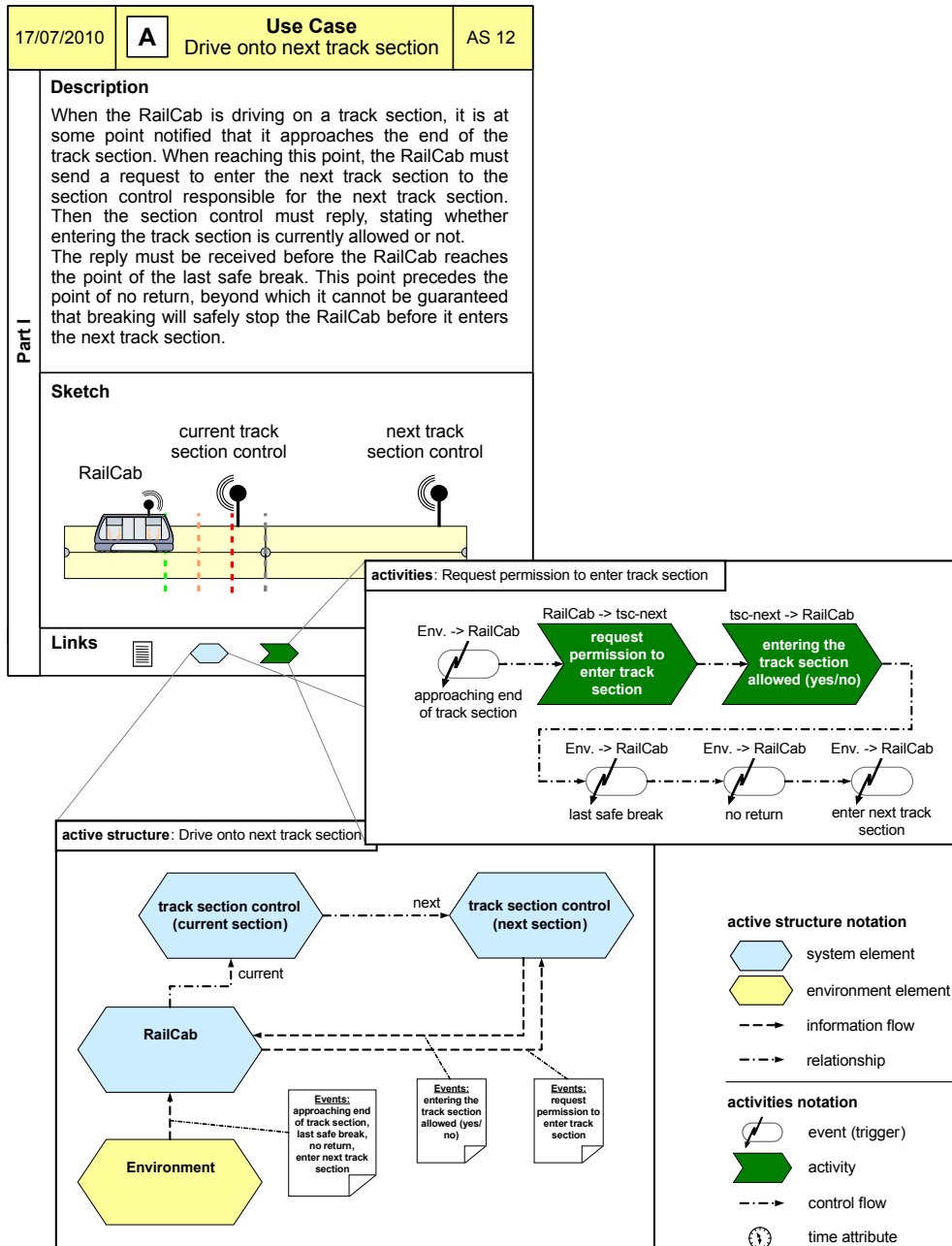


Figure 2.3: The informal description, active structure diagram and activity diagram for the use case Drive onto track section

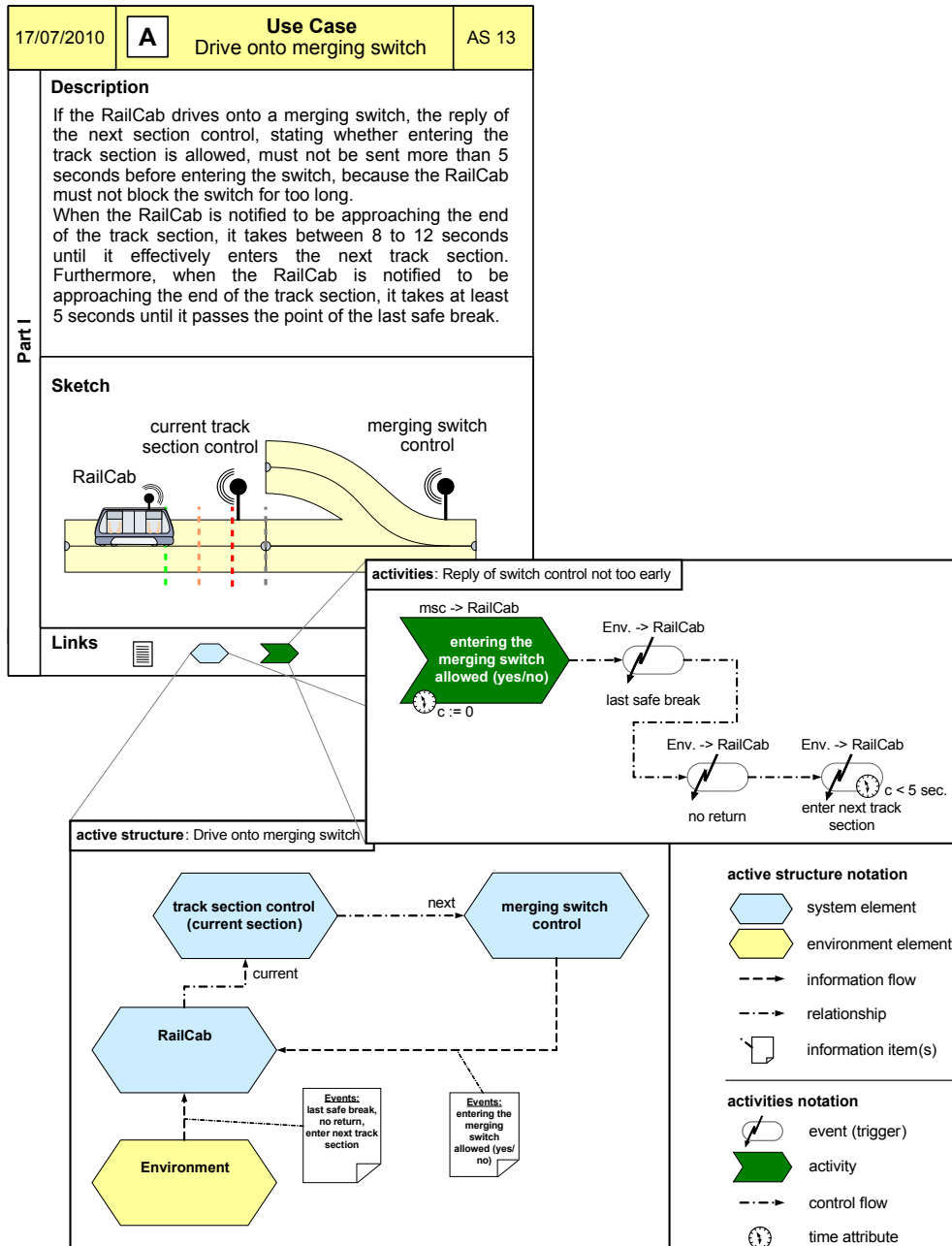


Figure 2.4: The informal description, active structure diagram and activity diagram for the use case Drive onto merging switch

`switch allowed` the same as the action `entering the track section allowed`? Even if the actions were called identically, it is not defined if this implies a relationship between the diagrams. Apparently, the merging switch control is a special kind of track section control and that should also exhibit all the behavior specified for the track section control, but this is not explicitly specified anywhere.

What can happen? What must happen? The second problem is that it is not clear whether the above activities describe possible scenarios or a behavior that must always happen. For example, it seems that a RailCab must request the permission to enter the next track section *whenever* it approaches the end of its current track section. However, not each event seems to be strictly required. For example, is it required that the events `last break`, `no return`, `enter next track section` must occur in this order? It seems that for describing the behavior of use cases, concepts are missing for describing whether a certain event or a sequence of events must happen or not. Furthermore, if there is a required sequence of events, it must be possible to describe when this sequence is required to occur, for example after a certain event or sequence of events.

What is the behavior of the system's environment? The third problem is that it is not clear what happens in the environment of the system. For example, the informal description of the use case `Drive onto merging switch` contains information about time intervals between points that the RailCab passes on the track section. This information is not contained in the activity diagram, but it is important in order to consider this information in order to determine whether the RailCab can fulfill the required time constraints.

Not only are these diagrams imprecise. Depending on the interpretation of the use cases, the requirements posed by the use cases may be *inconsistent*. Fig. 2.5 illustrates the system behavior and the assumed time intervals between the environment events as they are informally described in the above use cases. It is required that the switch control sends the reply `entering merging switch allowed` (abbreviated `enterAllowed`) before the RailCab passes the point of the last safe break (abbreviated `lastBreak`), but not more than 5 seconds before the RailCab effectively enters the next track section (abbreviated `enterNext`). If we do not assume any maximal time after which the RailCab will enter the next track section, it would not be possible at all to ensure that the track section control can adhere to the time requirements.

The informal description of the use case `Drive onto merging switch` formulates assumptions about the time intervals between certain environment events. These are illustrated at the bottom of Fig. 2.5. The event `lastBreak` is assumed to occur at least 5 seconds after the RailCab is notified about approaching the end of the track section (abbreviated `endOfTS`). The event `enterNext` is assumed to occur 8 to 12 seconds after `endOfTS`.

With these assumed time intervals, it turns out that it is actually not always possible for the system to adhere to the specified time constraints: The system must assume that `lastBreak` occurs already 5 seconds after `endOfTS`; therefore,

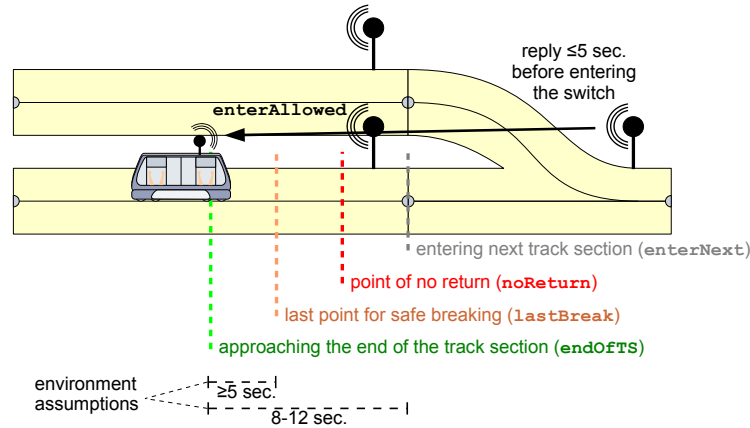


Figure 2.5: An illustration of the time constraints formulated in the “Drive onto merging switch” use case.

it will be necessary for the track section control to send the reply `enterAllowed` less than 5 seconds after `endOfTS`. But then, if it takes up to 12 seconds for the event `enterNext` to occur after `endOfTS` occurred, more than 7 seconds may elapse between `enterAllowed` and `enterNext`. This violates the requirement that not more than 5 seconds must pass between `enterAllowed` and `enterNext`. Therefore the specification is *inconsistent*, because no system will be able to always satisfy the requirements. The inconsistency could be resolved by either extending the maximal time from `enterAllowed` to `enterNext` to 8 seconds, or by changing the environment assumptions.

In summary, the existing design methodology must be extended by an adequate language for precisely specifying scenarios. Furthermore, it must be possible to automatically analyze the scenarios for inconsistencies. In the following, let us briefly overview other existing scenario-based design approaches.

2.4 Existing scenario-based design techniques

In software engineering, scenario-based design techniques are commonly applied to master the increasing complexity of software systems. One of the first languages to describe scenarios were *Message Sequence Charts* (MSCs), which originated in the telecommunication domain [ITU96], and were later included as sequence diagrams in UML [UML09]. An MSC describes a possible interaction scenario between a set of objects in the regarded system. Possible scenarios are also called *existential* scenarios in the following. The objects are represented by vertical lines, called *lifelines*, and arrows represent the exchange of *messages* between the objects. The vertical axis denotes the temporal order among the exchanged messages.

To express the relationship between single scenarios, *high-level MSCs* (hMSCs) were introduced. hMSCs express that there is a certain order among single MSCs or that certain MSCs represent alternative scenarios. There were soon efforts to use MSCs and hMSCs for not only documentation purposes. Krüger

for example presented a technique for creating state machines from alternative scenarios [KGSB99] and other kinds of hMSCs [Krü00]. Uchitel et al. presented a technique for translating alternative MSCs to labeled transition systems (LTS) [UK01, UKM03]. Similar techniques were presented by Leue et al. [LMR98] and Borderlau et al. [BCS00].

These approaches, however, did not consider the overlapping of scenarios. Overlapping scenarios were for example considered by Koskimies et al. [KSTM98] and by Maier and Zündorf [MZ03], who propose a technique for synthesizing statecharts from overlapping scenarios. The scenarios in these approaches, however, were all considered to be existential scenarios and could not express that something must or must not happen. Whittle and Schumann later extended existential scenarios by annotating messages in the MSC with pre- and post-conditions [WS00, WS02]. They show how conflicts arise when different MSCs specify contradicting conditions after the same sequence of messages. Such an approach, however, is only adequate if the engineers can provide enough meaningful pre- and post-conditions during the early design. This is, however, often not the case.

Within the SCEBASy approach, developed at the Software Engineering Group in Paderborn [Gie03, GB04, GT05, GKB05, GHHK06, HGH⁺09], UML sequence diagrams were extended with time annotations. Furthermore, mandatory sequences of events could be specified in the sequence diagrams. To express the relationships among the sequence diagrams, the lifelines must be annotated with states that the represented object is in after sending or receiving a certain message. Inconsistencies can occur when contradicting timing requirements are specified or when there are different messages in two sequence diagrams following the same state in mandatory sequences. If there are no contradictions, Real-Time Statecharts [GB03, BG03] for the interacting objects can be synthesized from the scenarios that realize the specified interaction behavior. The disadvantage of this approach is, however, that it requires the engineer to define the states of the interacting objects, which is, similar to the pre- and post-conditions above, information that is typically not available in the early design.

Because state annotations are difficult to provide, Damas et al. propose to specify use cases by a combination of existential and negative (forbidden) scenarios and properties expressed in temporal logic [DLDvL05, DLvL06]. They present an interactive approach where an algorithm generates questions when uncertainties or conflicts are detected in the scenarios. After the engineer answers these questions, a state-based implementation of the specified behavior is synthesized. It may be, nevertheless, that inconsistencies remain, especially if the user answers the questions inconsistently. Similarly, Uchitel et al. [UBC07, UBC09, SUB08] propose to specify interactions by a combination of existential scenarios and safety properties temporal logic. They also propose a technique for synthesizing a state-based implementation from the interaction specification. In this process, inconsistencies manifest in the form of deadlocks, which can be found in a subsequent analysis step. The disadvantage with these approaches is, however, that the proposed languages do not distinguish between events that may happen and events that must happen.

Addressing this problem, Damm and Harel extended MSCs to *Live Sequence Charts* (LSCs). In LSCs, existential scenarios can be described as well as scenarios that formulate constraints over all interactions in the system. Furthermore, it is possible to describe that, after a particular sequence of events, certain events can, must or must not occur. Techniques have been proposed to find inconsistencies in LSC specifications and for deriving state-based implementations from an LSC specification. [HK99, HK02, BS03, BSL04, BH05, KPP09, KS09]. Very useful, furthermore, is the *play-out* algorithm, developed by Harel and Marelly [HM02a, HM03]. This algorithm can execute certain kinds of LSCs, permitting the early simulation of the specified behavior.

Recently, Harel and Maoz proposed *Modal Sequence Diagrams* (MSDs) [HM08], a formal interpretation of UML sequence diagrams based on the concepts of LSCs. MSDs simplify some of the LSC language constructs. Due to the language features as well as the existing analysis techniques, MSDs are the most adequate basis for the development of a new technique for the scenario-based design of mechatronic systems. However, thus far no concepts have been presented for specifying environment assumptions with MSDs/LSCs and the existing synthesis and play-out techniques are not sufficient for finding inconsistencies in MSD/LSC specifications with real-time constraints and environment assumptions. (A more detailed discussion on the related work follows in Chap. 8.)

Foundations

This chapter introduces the fundamental concepts, languages and tools used in this thesis. Modal Sequence Diagrams (MSDs) are introduced in Sect. 3.1. Next, Sect. 3.2 introduces Timed Game Automata (TGA) and the game solving algorithm of UPPAAL TIGA, which forms the basis of the synthesis technique elaborated in this thesis. Last, Sect. 3.3 explains the basics of Triple Graph Grammars (TGGs) and TGG-based model transformations.

3.1 Modal Sequence Diagrams

This section introduces Modal Sequence Diagrams (MSDs), the structure of an MSD specification, and its semantics. It also explains different notions of consistency and the principles of the play-out algorithm. The definitions in this section are mainly based on the definitions of LSCs by Harel and Marelly [HM03] and the definitions of MSDs by Harel and Maoz [HM08, Mao09]. However, some definitions are also modifications or extensions to the original definitions. Within the scope of this thesis, especially a variant of the play-out algorithm was implemented in the SCENARIOTOOLS tool suite; this algorithm has some differences to the original algorithm [HM03].

3.1.1 MSDs and the object system

A basic MSD specification consists of a set of MSDs where each lifeline represents exactly one object in a fixed *object system* [HM03]. The objects in the system can send and receive *messages*. The middle of Figure 3.1 shows an example of an object system. It consists of the objects `env:Environment`, `rc:RailCab` and `tsc:TrackSectionControl`. Two MSDs `RequestEnterAtEndOfTrackSection` and `ReplyBeforeLastSafeBreak` that make up the MSD specification are displayed at the bottom. Which lifeline represents which object is indicated by the label of the lifeline, and, in this figure, is additionally indicated by the dashed lines.

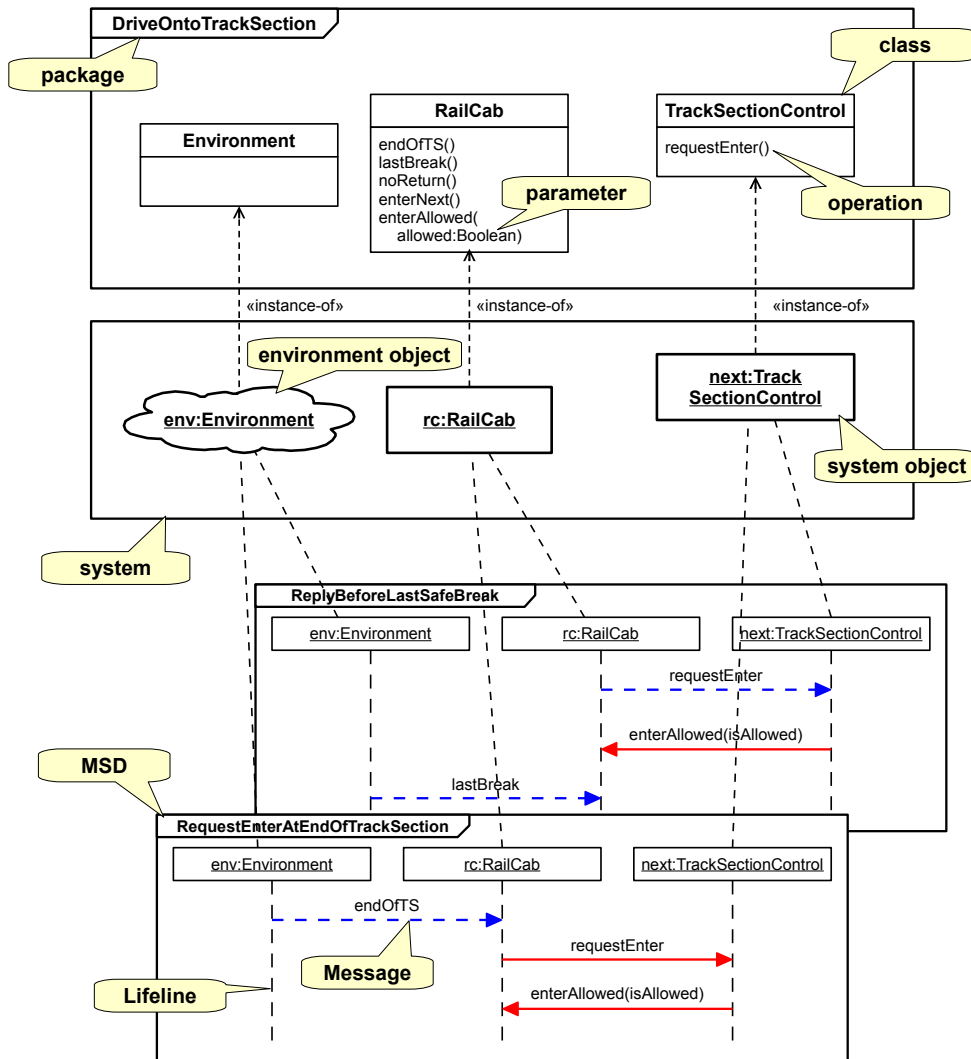


Figure 3.1: The object system and MSDs with concrete lifelines

An MSD specification subdivides the objects in the system into *environment objects* and *system objects*. Environment objects are displayed as cloud-like shapes; system objects are displayed as rectangles. The set of system objects is also called the *system*, the set of environment object is also called the *environment*. The objects in the system are instances of *classes* that are contained in a *package*. The top of Fig. 3.1 shows the package `DriveOntoTrackSection` with the classes `Environment`, `RailCab` and `TrackSectionControl`. A class can have *operations* that specify which messages an object that is an instance of that class can receive. A message is identified by the sending object, the receiving object and an operation defined by the class of the receiving object. An operation has a list of *parameters* that are typed over *data types*, which may be Boolean, Integer, or other data types. A message sent between two objects carries a list of values for each parameter of the operation that the message refers to. In the figure,

the class `RailCab` has a number of operations. The operation `enterAllowed`, for example, has the Boolean parameter `allowed`.

Intuitively, the MSDs shown above express the following requirements. The MSD `RequestEnterAtEndOfTrackSection` says that if the `RailCab` receives the message `endOfTS` from the environment, the `RailCab` must send the message `requestEnter` to the next track section control. The track section control must then send the message `enterAllowed` back to the `RailCab`. The parameter of the message is not specified in this case. The MSD `ReplyBeforeLastSafeBreak` says that if the `RailCab` sends the message `requestEnter` to the track section control, the track section control must send the message `enterAllowed` back to the `RailCab`. Again, the parameter value for the message is not specified. After the `RailCab` receives this message, the environment can send the message `lastBreak` to the `RailCab`. This message must, however, not occur before the message `enterAllowed` was sent. This behavior is essentially what was informally described in the use case `Drive onto track section` shown in Fig. 2.3. The constructs in the MSDs and their semantics are explained in more detail in the following.

3.1.2 Events, messages, and runs

Whenever an object in the object system sends or receives a message, this is called an *event*. For simplicity, only *synchronous* messages are regarded in the scope of this thesis, where the sending and receiving of a message happen synchronously. The sending and receiving of a message are therefore also regarded as one single event. Messages that are sent from environment objects are called *environment messages* or *environment events*, messages sent from system objects are called *system messages* or *system events*. An infinite sequence of events in the object system, in other words, an infinite sequence of message interchanges between the objects, is called a *run* of the object system. Here we regard infinite runs, because we consider systems in which the system objects react to sequences of environment events with arbitrary lengths, systems that may run “forever”.

3.1.3 Event unification

A *message* in an MSD, for clarity also called a *diagram message*, is depicted as an arrow between two lifelines. A diagram message represents an event in the object system. More specifically, a message in an MSD represents an event in the object system if the sending lifeline in the MSD represents the sending object, if the receiving lifeline in the MSD represents the receiving object, and if the message in the diagram refers to the same operation as the message sent in the object system. If that is the case, a diagram message is also said to be *unifiable* with an event occurring in the object system. It will be explained shortly how the concept of event unification is extended in the case of parameterized messages.

3.1.4 Existential and universal MSDs, hot and cold messages

An MSD describes a partial order among events in the object system. There are two kinds of MSDs. First, there are *existential* MSDs, which describe an order of events that must be possible to occur in at least one run of the object system. Second, there are the *universal* MSDs, which formulate requirements for all possible runs of the object system. Within the scope of this thesis, only universal MSDs are regarded.

The messages in a universal MSD have a *temperature* and an *execution kind*. The temperature of a message can be either *hot* or *cold*. Hot messages are colored red; cold messages are represented by blue arrows. The execution kind of a message can be either *executed* or *monitored*. Executed messages have a solid line; monitored messages have a dashed line. The first message in an MSD is always cold and monitored. Each universal MSD *accepts* a set of runs of an object system. What the temperature and the execution kind mean and when a run is accepted by an MSD is explained in the following.

3.1.5 Active MSDs, the cut, and hot and cold violations

Messages that have no preceding message in an MSD are called a *minimal event*. The MSDs regarded in this thesis are required to only have one minimal event, which is therefore also called the *first event*. If in a run in the object system an event occurs which is unifiable with this first event, an *active copy* of the MSD, or *active MSD* in short, is created. On the occurrence of further events in the run that are unifiable with the subsequent messages, the active MSD progresses.

To capture this progress, the passed *locations* on each lifeline are marked. Locations are the points on each lifeline where the arrows of the diagram messages are attached. The set of already passed locations in the active MSD copy is called the current *cut* of the active MSD. If the cut reaches the end of the diagram, the active copy of the MSD is discarded.

If the cut of an active MSD copy is immediately before a message on the sending and receiving lifeline, the message is *enabled*. If one of the enabled messages in an active MSD is hot, the cut is *hot*. If all enabled messages are cold, the cut is *cold*. Similarly, if one of the enabled messages in an active MSD is executed, the cut is *executed*. If all enabled messages are monitored, the cut is *monitored*. Enabled executed messages are also called *active messages* or *active events*.

If the active MSD copy is in a hot cut, it is not allowed for events to occur in the object system which are unifiable with a message in the MSD that is not currently enabled. If such an event occurs, this is a *safety violation* or *hot violation* of the MSD. An MSD does not accept any run in which a safety violation occurs. If the active MSD copy is in a cold cut and an event occurs that is unifiable with a message in the MSD that is not currently enabled, this is a *cold violation* of the MSD. Cold violations are allowed to occur, but if a cold violation occurs, the active copy of the MSD is discarded.

If the cut is executed, but never progresses, this is also called a *liveness violation* of the MSD. An MSD does not accept any run in which a liveness

violation occurs. If an active MSD is in a monitored cut, by contrast, it may remain in this cut forever.

The semantics of the message temperature and the execution kind in this thesis differ slightly from the MSD semantics defined by Harel and Maoz [HM08]. Originally, the semantics of a hot message is also that, if enabled, it must be eventually progressed, i.e., if a hot message is enabled, a safety and a liveness violation may occur. In this thesis, however, the above semantics of the temperature and the execution kind is defined as described above in order to be able to differentiate safety from liveness requirements. A hot and executed message in the above semantics has the same meaning as a hot message in the original semantics.

3.1.6 The iterative interpretation of MSDs

If the first message or the first messages in an MSD occur repeatedly later in the MSD, it is possible for multiple active copies of the MSD to be created. There are, however, approaches where such a behavior is explicitly excluded [HK02, BS03, BSL04, BH05, KTWW06] and a new active copy of an MSD is only created if currently no active copy of that MSD exists. This is called the *iterative* interpretation. If multiple copies are allowed, as explained above, this is called the *invariant* interpretation. The iterative semantics allows for an easier analysis of the MSDs and the limitation that this interpretation implies is only relevant in a few cases. The synthesis technique that was elaborated in the scope of this thesis is based in the iterative interpretation of MSDs, which will be explained in the following in more detail.

In the iterative interpretation of MSDs, the runs that are accepted by each universal MSD can be represented by a *Büchi automaton* [Büc66]. A Büchi automaton is a finite state automaton that accepts infinite words. The states in a Büchi automaton are partitioned in *accepting* states, represented by a double circle, and *rejecting* states, represented by a simple circle. Furthermore, a Büchi automaton has a start state, represented by an incoming arrow. Between the states, a Büchi automaton has transitions, labeled by symbols of an alphabet. A Büchi automaton accepts an infinite word, which is an infinite sequence of symbols from the alphabet, if there exists a sequence of transitions starting from the start state and visiting accepting states infinitely often, such that the sequence of transitions corresponds to the sequence of symbols in the given infinite word. The alphabet in the following will be the set of possible events in the object system. A Büchi automaton for a universal MSD is constructed as follows, inspired by the automata construction proposed by Bontemps [BH02, BS03] and Harel et al. [HM08].

Consider the MSD in Fig. 3.2. This MSD is a variant of the MSD Request-EnterAtEndOfTrackSection of Fig. 3.1. In this MSD, the message `enterAllowed` is hot and monitored and, additionally, the cold monitored message `lastBreak` is appended in this MSD. The dashed horizontal lines show the different states that an MSD can be in according to the iterative semantics. The line labeled `s0` represents the state where the MSD is inactive, the lines `s1`, `s2`, and `s3` represent the different reachable cuts when the MSD is active. The state `s0` is

also called the *initial cut* of the MSD. The initial cut and all reachable cuts are called the *legal cuts* of an MSD.

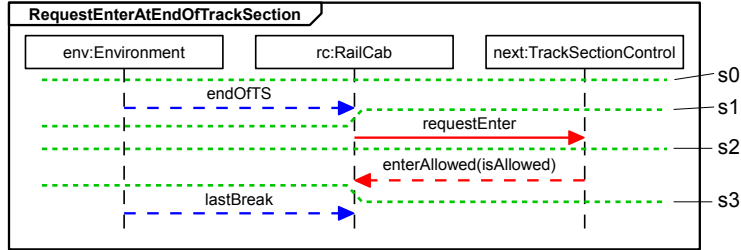


Figure 3.2: An extended version of the MSD RequestEnterAtEndOfTrackSection

A Büchi automaton for a universal MSD consists of one state per legal cut in the MSD and a rejecting state, r . The states representing legal cuts are also called *cut states*. The state r represents the situation where a safety violation occurred in the MSD. The state representing the initial cut is the start state of the Büchi automaton. The state representing the initial cut and each state that represents a monitored cut is an accepting state. Figure 3.3 shows the Büchi automaton created for the MSD RequestEnterAtEndOfTrackSection in Fig. 3.2. It consists of the states s_0, s_1, s_2, s_3 , and r . Since the state s_0 represents the initial cut, and the states s_2 and s_3 represent monitored cuts, these states are accepting states.

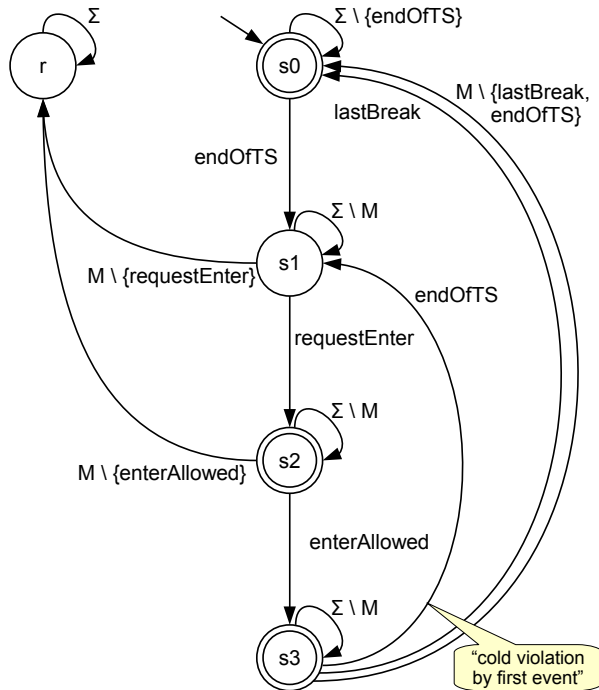


Figure 3.3: The Büchi automaton corresponding to the iterative interpretation of the universal MSD RequestEnterAtEndOfTrackSection in Fig. 3.2.

Between these states, transitions are created as follows. First, transitions are created from each state that represents a legal cut to each following legal cut. These transitions are labeled with the events that progress the (active) MSD from the cut represented by the source state to the cut represented by the target state. Figure 3.3 shows the transition labeled by the event `entOfTS` from `s0` to `s1`, the transition labeled by the event `requestEnter` from `s1` to `s2`, and the transition labeled by the event `enterAllowed` from `s2` to `s3`. Additionally, from states that represent cuts without a following cut, transitions are created back to the state representing the initial cut. These transitions are labeled with the events that terminate the MSD in the cut represented by the source state. In Fig. 3.3, this is the transition labeled with the event `lastBreak` from state `s3` to state `s0`.

Second, each state in the Büchi automaton has a self-transition. These self-transitions are labeled with sets of events. The self-transition on `r` is labeled with all events that may occur in the object system. This set is called Σ in the following. Intuitively, this means that, if a safety violation has occurred, the automation will loop in this rejecting state forever, thus rejecting the run. The self-transition on the state representing the initial cut is labeled with the events Σ without the first event of the MSD, which is `endOfTS` in the above case. Intuitively, this means that the MSD, when inactive, ignores all events except the first event; in this state the automaton would for example accept any sequence of events in which the first event of the MSD never occurs. The remaining states, which represent the different reachable cuts of the active MSD, have a self-transition for all events Σ without the events occurring in the MSD. The latter set is called M . Intuitively, this means that the active MSD ignores all events that are not occurring in the MSD. An active MSD remaining in a monitored cut forever or an MSD that is always inactive accepts the run.

Last, there are transitions in the automaton which represent safety violations and cold violations. Each state that represents a hot cut has a transition to the state `r`. This transition is labeled with all the events that are appearing in the MSD without the events that are enabled in this cut. In the example, the cuts `s1` and `s2` are hot and transitions labeled $M \setminus \{\text{requestEnter}\}$ and $M \setminus \{\text{enterAllowed}\}$, respectively, lead from these states to `r`.

Each state that represents a cold cut has two transitions which represent two different cases of cold violations. The first case is the cold violation by the first event in the MSD. This transition leads to the state that represents the first reachable cut, `s1` in the above case, and is labeled by the first event. Intuitively, this transition represents the behavior where the active copy of the MSD is discarded by a cold violation, but immediately another active copy of the MSD is created. This transition is omitted only in cut states where a message is enabled that represents the same event as the first message in the diagram. (The events may again be represented by another message later in the MSD.)

The second transition represents the case where the cold violation is caused by an event that is represented by a message in the MSD, but that is not represented by the first message in the MSD. This transition leads to the state representing the initial cut and is labeled by the events that are not currently enabled without the first event. The transition in the above example from `s3`

to s_0 , labeled with the events $M \setminus \{\text{lastBreak}, \text{endOfTS}\}$, represents the cold violation by an event that is not the first event in the MSD.

3.1.7 Satisfying an MSD specification

For the set of system objects, we can specify a *controller* that determines which message and object sends after a certain sequence of events in the object system. This controller can be non-deterministic. The environment objects are uncontrollable, or, in other words, the environment objects could be controlled by any possible controller. If there is no concrete controller for the environment objects, the system is also called an *open system* [HP85]. If there is a controller for the environment objects, the composition of the environment controller and the system controller forms a controller for the whole object system. Such a system is also called a *closed system*.

A closed system *satisfies* an MSD specification, consisting of a set of universal MSDs, if all runs of the system are accepted by all universal MSDs in the specification. An open system *satisfies* an MSD specification if the closed system formed for all possible controller for the environment satisfies the MSD specification. A controller of an open system that satisfies an MSD specification is also called an *admissible controller* or *admissible implementation* of the specification

We assume a setting where the sending of messages takes no time and the system is able to send any number of messages before the next environment event occurs. This assumption is also called the *synchrony hypothesis* [BB91]. However, we do not desire any system that at some point reacts with an infinite sequence of system events. Therefore, as also stated by Kugler et al. [KPP09], for a system to satisfy an MSD specification, it is additionally required that there must not be any run with an infinite sequence of system events. In other words, the system must “listen” to environment events infinitely often.

3.1.8 The play-out algorithm

The play-out algorithm is an operational interpretation for universal LSCs and MSDs, originally defined by Harel and Marelly for LSCs [HM02a, HM03]. The idea is that, instead of implementing a controller for the system objects, we just interpret the MSDs/LSCs, which tell us which messages the system objects can, must, or must not send after a certain sequence of events. The play-out of universal MSDs works as follows.

First, the system does nothing and waits for environment events to occur. Then, just as described in Sect. 3.1.5, active copies of MSDs are created if an environment event is unifiable with the first message in the respective MSDs. Next, if there are active MSDs that reside in executed cuts, there is a set of active (enabled and executed) events postulated by these active MSDs. The play-out algorithm then non-deterministically chooses to execute one of these events, i.e., orders the respective object to send the respective message, if this does not lead to a safety violation of any other active MSD. This progresses the cut of some active MSDs and may cause the activation of further MSDs, which

then again postulate a set of active events from which the play-out algorithm chooses the next to execute. This process is repeated until there is no active MSD with an executed cut left. Then the algorithm again listens for the next environment event. The sending of a message is also called a *step* in the play-out algorithm. This sequence of steps until the algorithm listens for the next environment event is also called a *super-step*.

The play-out algorithm can of course only order system objects to send messages. Typically, environment messages will not be marked as executed. However, if there is an executed environment message that is enabled at some point, the system is at the environment’s mercy to eventually send an according message. Otherwise it will not be able to satisfy the MSD specification. Only if it is a cold executed message, the system may be able to cause a cold violation of the respective active MSD and avoid to stay in an executed cut forever.

It may happen that the play-out algorithm runs into a situation where there is a set of active events, but all of these events are forbidden to occur, because they would lead to a safety violation of another MSD. Then the play-out algorithm cannot process. This situation is also called a *hot violation*. Sometimes, if a hot violation occurs, it could have been avoided if the play-out algorithm had chosen to execute another message at some point in the past, or, in other words, if it would have “looked ahead” earlier to see which sequence of steps will avoid a hot violation. The play-out algorithm can, however, not look ahead and thus may easily run into hot violations.

To alleviate this problem, Harel et al. have developed an improved play-out algorithm, called *smart play-out* [HKMP02a]. Smart play-out can look ahead one super-step and can thus avoid some avoidable violations. It can, however, not avoid all violations, because the decisions of the algorithm in one super-step can still make a hot violation in a future super-step inevitable.

Figure 3.4 illustrates different executions of a play-out algorithm that can lead to a hot violation (red X) or a valid super-step (green check symbol). The clouds on the left with the dashed arrows represent the environment events that occur and the wiggly black lines are alternative super-steps that the play-out algorithm can take in response to an environment event. The black dot at the top represents the initial state of the play-out.

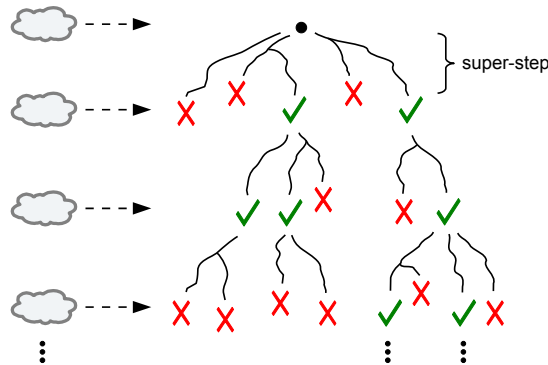


Figure 3.4: An illustration of valid and violating super-steps.

Even though there is one sequence of super-steps on the right by which the play-out algorithm could avoid hot violations, the smart play-out algorithm may choose to execute another valid super-step on the left and then inevitably run into a hot violation in the third super-step.

3.1.9 Consistency, consistent executability, and realizability

An MSD specification is *consistent* if there exists an admissible controller for the MSD specification, or, in other words, if there exists a controller for the system that, if composed with any possible controller for the environment, satisfies the MSD specification [BH05]. A consistent specification is also said to be *realizable*.

A specification is *consistently executable* if the play-out algorithm could satisfy the MSD specification. Then we also say that there exists an *admissible strategy* for the play-out algorithm. Kugler et al. [KPP09] also call this *non-violating execution*. There are two differences between an admissible strategy of the play-out algorithm and a general controller. First, the play-out algorithm can only execute active events, i.e., it can only send messages if they are represented by an executed message that is currently enabled in an active MSD. Second, the play-out algorithm will always terminate its current super-step before listening to the next environment event.

Note that consistent executability is a stronger property than consistency. Even if there exists no execution of the play-out algorithm that satisfies the specification, the specification may still be consistent. For example, if a play-out algorithm cannot avoid a hot violation from some point on, it may be possible to modify the play-out algorithm so that it can at some point execute an event that is not currently active. By this modification, it may be possible for the play-out algorithm to induce a cold violation in an active MSD that would otherwise lead to the occurrence of a hot violation later on. The same is true if the play-out algorithm would be modified to not necessarily terminate every super-step before the next environment event occurs. In rare cases it may be possible that this way an environment event leads to a cold violation of an MSD that would be otherwise accountable for a hot violation.

Due to the limitations of play-out and smart play-out, which cannot always avoid avoidable violation, it is not possible to use these techniques for determining whether an MSD specification is consistent or consistently executable. For this purpose, a synthesis technique is elaborated in this thesis (see Chap. 4). This technique is able to decide both the consistency and consistent executability of an MSD specification.

Typically, the property that we want to show for an MSD specification is consistent executability: an engineer wants to know “does there exist a system that—by doing what it should or must do—and by not doing anything more—can satisfy the requirements?” Sometimes, however, an engineer is rather interested in the consistency of the MSD specification: then an engineer wants to know “does there exist a system that by doing everything it can do—even doing what we did not explicitly order it to do—including even doing nothing—can satisfy the requirements?” An MSD specification that is not consistently executable can become consistently executable by adding further MSDs. By

contrast, an MSD specification that is not consistent cannot become consistent or consistently executable by adding further MSDs. Therefore, the engineer may want to know whether an MSD specification is consistent if he wants to know whether an existing specification is a consistent basis on which to add further MSDs.

3.1.10 Parameterized messages

Some of the MSDs introduced previously contain messages with parameter (see for example the message `enterAllowed(isAllowed)` in the MSD `RequestEnterAtEndOfTrackSection` in Fig. 3.1). A message in the diagram must either provide concrete values for the parameters of the operation that the diagram message refers to, or it must specify variables for the parameter, or a combination of variables and literal values. The variables have a type that must be equal to the type of the parameter. In an active MSD, a variable can be *unbound* and it can become *bound* to a value of that type as explained shortly. These variables are only visible within the scope of an active MSD; if the active MSD is terminated, the variable is undefined. These variables are also called *diagram variables*.

In Sect. 3.1.3, it was defined that a diagram message is *unifiable* with an event in the object system if the sending and receiving objects of the event are represented by the sending and receiving lifeline of the diagram message, and the event in the object system and the diagram message refer to the same operation. For parameterized messages, this concept must be extended. If an event and a diagram message are unifiable in the above sense, but the message is parameterized, we say that they are *message unifiable*, but they are not yet necessarily *unifiable*. In the parameterized case, the diagram message must furthermore, for each parameter of the message, either specify the same literal value, a variable bound to the same value, or an unbound variable. If that is the case, the diagram message is also said to be *parameter unifiable* with the event in the object system. A parameterized diagram message is only *unifiable* with an event in the object system if it is both message and parameter unifiable to that event.

The semantics of parameterized messages in MSDs is the following. If a parameterized message is enabled, and a message occurs in the object system that is unifiable with the enabled diagram message, the cut progresses beyond this diagram message. If, however, an event occurs that is message unifiable with an enabled message or another message in the MSD, but not unifiable, i.e., message and parameter unifiable, with an enabled event, this is a violation of the diagram. As before, it is a forbidden safety violation if the cut is hot, and it is an admissible cold violation if the cut is cold.

In the process of unifying an event in the object system with a diagram message, all unbound variables are bound to the values carried by the event in the object system.

If during play-out a parameterized message is active where no concrete values for the parameters are specified, the play-out algorithm can non-deterministically choose values for these parameters. The chosen values must, however, not lead to a safety violation in any active MSD. More precisely, if

there are other hot messages enabled that are message unifiable with the executed message, the values must be chosen such that the executed message is also parameter unifiable with these enabled messages. If there exists no such assignment of message parameters, it is a hot violation.

3.1.11 Object properties and side-effects

Classes can define properties that have a name and a certain data type, like Integer, Boolean, or others. Instances of these classes carry concrete values for these properties. These property values can be read in MSDs and an MSD can specify how such properties can or must be modified.

Reading object properties

Consider a scenario that requires that the request of a RailCab to enter the next track section must be refused by the track section control if a hazard has occurred on the next track section. In other words, the track section control must answer with `enterAllowed(false)` when the RailCab sends the message `requestEnter`, but a hazard has occurred on the next track section. Let us assume that whether a hazard has occurred on a track section or not (or has been resolved again) is stored in the property `hazardOccurred` of the track section control. In the MSD, we can then specify a value for a message parameter. For this purpose, we use OCL expressions [OCL10] in the scope of this thesis. In these expressions, diagram variables can be used, but also the lifeline names can be used as variables. These variables, called *lifeline variables*, are bound to the object represented by the lifeline. To specify message parameters, expressions can be used that evaluate to a value that has the same type as the specified parameter. Figure 3.5 shows the MSD `EnterDeniedWhenHazardOccurred` where the above requirement is formalized. Here the expression `next.hazardOccurred` is specified as the parameter for the message `enterAllowed`. This expression evaluates to the value of `hazardOccurred` property of the object represented by the next lifeline.

In such a case, events in the object system are only unifiable with this diagram message if the value of the parameter equals the evaluation of the expression in the diagram. In this example, it means that if the RailCab requests the permission to enter the next track section, and the property `hazardOccurred` is set to true, the track section control must not allow the RailCab to enter. If `hazardOccurred` is set to false, the track section control must allow the RailCab to enter.

Side-effects

Instead of just reading object properties, we would also like to specify that certain values should or must be assigned to object properties. This can be done by *side-effects* of messages. Harel and Marelly define that a parameterized message (with one parameter) can be associated with the property defined for the receiving object. Upon an occurrence of that message, the parameter value

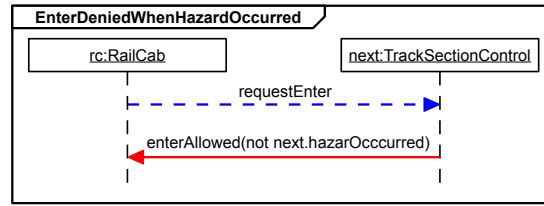


Figure 3.5: A message parameter specified by an expression over an object property

carried by the message occurrence is assigned to the object's property. [HM03, Sect. 5.6].

Within this thesis, the convention is used that if a class defines a property and also defines an operation `set<PropertyName>(newValue:<PropertyType>)`, events in the system that are typed by this operation assign the carried parameter value to the according property of the receiving object. By referring to these messages, we can then specify in the MSDs how the object properties can or must be modified. Figure 3.6 shows an example: if the RailCab detects an obstacle, it must then set the attribute `hazardOccurred` of the current track section control to true. This convention is also implemented in the SCENARIOTOOLS play-out algorithm.

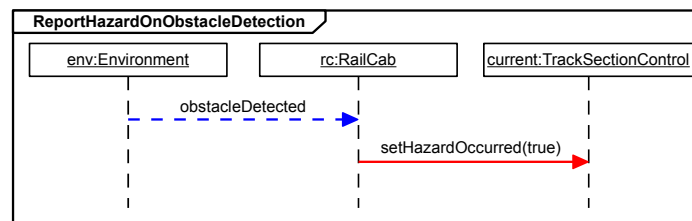


Figure 3.6: An example for a message with a side-effect on the receiving object

3.1.12 Assignments and conditions

An MSD may also contain *assignments* and *conditions*, which are displayed as boxes or hexagons, respectively, that can span one or multiple lifelines in the MSD. By conditions, it can be specified if and when the cut of an MSD progresses. Assignments can be used to modify diagram variables.

Conditions

Conditions have a *condition expression* and a temperature. Hot conditions are displayed as hexagons with a solid red border. Cold conditions are displayed as hexagons with a dashed blue border. The condition expression is displayed inside the hexagon. Where a condition spans lifelines, these lifelines also have *locations* and a condition is *enabled* if the cut is immediately before the condition's locations on all lifelines that the condition spans. Similar to the parameter

expressions above, a condition expression can be an OCL expression that may refer to diagram and lifeline variables. In the case of a condition, the expression must evaluate to a Boolean value.

The semantics of a cold condition is that if it is enabled, it is immediately evaluated, before another event occurs in the object system. If the expression evaluates to false, this is a cold violation of the diagram. If it evaluates to true, the cut of the active MSD progresses.

The expression of a hot condition is also evaluated immediately and, if it evaluates to true, the cut of the active MSD progresses. But, if it evaluates to false, the cut cannot progress. As soon as the expression can be evaluated to true, the cut progresses, but if that is never the case, the cut remains in front of the hot condition forever. This is then a liveness violation of the MSD, i.e., the MSD rejects runs where a hot condition is enabled, but the cut never progresses.

Assignments

Similar to a condition, an assignment has an *assignment expression* and can cover one or multiple lifelines. Where the assignment covers a lifeline, the lifeline also has locations, and the assignment is *enabled* if the cut is immediately before the assignment on all lifelines that it covers. The assignment expressions are of the form `<diagramVar> = <expr>`, where `<diagramVar>` is a diagram variable and `<expr>` is the *value expression*. The value expression is an OCL expression that may refer to diagram and lifeline variables and evaluates to a value of the same type as the diagram variable.

If an assignment is enabled, the assignment expression is immediately executed, i.e., the diagram variable is assigned the value that results from the evaluation of the value expression, before any other event in the object system occurs.

The left of Fig. 3.7 shows an example of an MSD that contains an assignment and a cold condition. After an occurrence of the event `requestEnter`, the value of the property `hazardOccurred` of the object represented by the next lifeline is assigned to the diagram variable `hazardOcc`. Subsequently, this diagram variable is used in a cold condition. If `hazardOcc` is true, the cut in the active diagram progresses and eventually an `enterAllowed` must occur with the `isAllowed` parameter set to `false`. If `hazardOcc` is false, this is a cold violation and the diagram is discarded. To the right, an MSD is shown that specifies the same behavior, but where the cold condition directly refers to the property of the object represented by the lifeline next.

3.1.13 Visible and Hidden events

Executing an assignment and progressing beyond a condition has no effect that is visible within the object system. Remember that diagram variables are only visible within an active MSD. Therefore, executing an assignment and progressing beyond a condition is also called a *hidden event*. By contrast, the sending of a message in the object system is visible in the object system and is therefore also called a *visible event*.

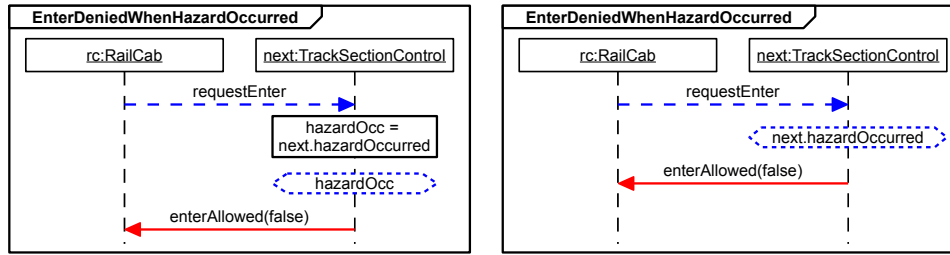


Figure 3.7: Simple examples of assignments and conditions in MSDs.

3.1.14 Timed MSD specifications

Time is an important aspect in mechatronic systems, where software controls physical processes. The use case Drive onto merging switch (see Fig. 2.4), for example, formulates time requirements in the RailCab system.

The notion of time was introduced to LSCs previously by Klose and Wittke [KW01]. Harel and Marelly similarly defined a discrete time concept for LSCs [HM02b, HM03]. Plock et al. introduced a *real-time* model to LSCs [PGZ05] where the passage of time is represented by a global real-valued *clock variable*. Assignments can assign the current value of that clock to *time stamp* variables and constraints can be formulated over the clock and time stamp variables. Larsen et al. [LLNP10] use multiple clock variables in LSCs that represent the passage of time. The time model used in this thesis is similar to the one of Larsen et al.

To represent time, *clock variables* are used in the MSDs. Clock variables are real-valued diagram variables that synchronously and monotonically increase in value over time. Clock variables can be reset to integer values in assignments, which we then call *clock resets*. We can refer to clock variables in certain kinds of conditions that we call *time conditions*. In time conditions, expressions can be of the form $x_1 \bowtie expr$ where x_1 is a clock variable, $expr$ is an expression without any clock variable, evaluating to an integer value, and \bowtie is an operator $\{<, \leq, >, \geq\}$.

Just as with regular conditions, enabled time conditions are evaluated immediately and the cut immediately progresses beyond the condition if the expression evaluates to true. If the expression of a cold time condition evaluates to false, this is a cold violation. If the expression of a hot time condition evaluates to false, the cut cannot progress. It progresses immediately as soon as the expression evaluates to true, but it is a liveness violation of the MSD if that is never the case. Hot time conditions that specify a lower time bound ($\bowtie \in \{>, \geq\}$) are also called *minimal delays*. Hot time conditions that specify an upper time bound ($\bowtie \in \{<, \leq\}$) are also called *maximal delays*.

Figure 3.8 shows the MSD ReplyOfSwitchControlNotTooEarly, which formalizes the requirement formulated in the use case Drive onto merging switch. The requirement says (see Fig. 2.4) that the reply `enterAllowed` of a merging switch control to the request to enter a merging switch must not be sent more than five seconds before entering the switch (`enterNext`). In the MSD, this time

constraint is described by the help of the clock variable c . After the event `enterAllowed` occurred, this clock variable is immediately reset to zero. Clock resets are displayed as boxes, like assignments, but with an hour glass symbol on the upper-right corner. After the message `enterNext`, a cold time condition checks whether the value of c is greater than five immediately after `enterNext` occurred. Time conditions are displayed as hexagons, like regular conditions, but also have an additional hour glass symbol on the upper-right corner.

If it takes more than five seconds from `enterAllowed` to `enterNext`, the cut progresses beyond the time condition. In this example, however, the cut will then be stuck before the terminal hot condition forever, because the expression will always evaluate to false. This is therefore a liveness violation of the MSD. An MSD of this form are also called an *anti-scenario*, because it describes a sequence of events that must not happen. If it takes exactly five seconds or less from `enterAllowed` to `enterNext`, there will be a cold violation of the time condition and the liveness violation of the MSD can be avoided. (For this example, we assume that the lifeline `next:MergingSwitchControl` refers to an object `next` in the object system, which is an instance of the class `MergingSwitchControl`, but we will not go into the details of that object system here.)

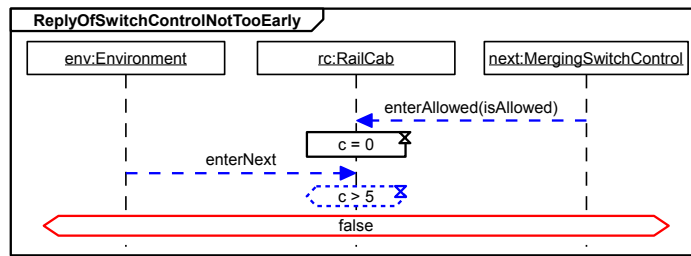


Figure 3.8: The MSD `ReplyOfSwitchControlNotTooEarly`.

Executing clock resets and evaluating time conditions are also hidden events. Also, as mentioned previously, it is assumed that events take no time and time can only elapse between events (*synchrony hypothesis*). But, any number of events may occur in zero time. As before, we assume that the system can take any number of steps in reaction to an environment event before the next environment event occurs. If, however, the system waits for time to elapse, environment events may occur. In order to *satisfy* a timed MSD specification, we require that no safety or liveness violation occurs in any MSD and that the system does not take infinitely many steps without waiting for environment events to occur or for time to elapse.

The play-out of timed MSD specifications works as the play-out of an un-timed MSD specification. The only difference is that the algorithm may have to wait to progress beyond minimal delays. Otherwise, active messages are executed immediately.

The directive of the play-out algorithm to always executes active messages immediately is a practical convention. However, for that convention, the play-out algorithm is unable to avoid certain kinds of violations in timed MSD specifications. For an example, consider a specification that consists of the MSDs in

Fig. 3.1 and the MSD above in Fig. 3.8. (We assume that there is some object system where the third lifelines in these MSDs represent the same object.) The play-out algorithm would send the messages `requestEnter` and `enterAllowed` immediately after `endOfTS` occurred. However, as discussed above for the MSD `ReplyOfSwitchControlNotTooEarly`, it is forbidden that more time elapses between `enterAllowed` and `enterNext`. Therefore, the immediate execution of an active message may lead to a violation that could be avoided by delaying the execution. However, because the play-out algorithm cannot look ahead, it cannot decide when delaying the execution of an active message is reasonable.

Smart play-out can look ahead in the scope of one super-step and, therefore, could decide to delay certain steps on the basis of the steps it can look ahead. This is, however, not implemented in smart play-out [HKP04]. Therefore, the smart play-out of timed MSD specifications may run into many avoidable violations. Even if smart play-out could look ahead, it would not be able to avoid the violation in the scenario sketched in the above paragraph. The reason is that the super-step where smart play-out could delay the `enterAllowed` message terminates after sending `enterAllowed`. Thus, the possible time of the occurrences of `lastBreak` cannot be anticipated by timed smart play-out. In the synthesis technique that will be presented in Chap. 4, we will consider this case.

3.1.15 Symbolic lifelines

The lifelines in the MSDs shown so far refer to concrete instances in the object system (see Fig. 3.1, Sect. 3.1.1). These lifelines are also called *concrete* lifelines. However, there are many systems, like the RailCab system, where just considering one object system is not sufficient. Instead, we wish that a specification applies to many different and possibly changing object systems. Then, we would like to describe how certain kinds of objects, which for example have certain relationships to each other, shall behave.

For example, we would like to specify that when any RailCab in the RailCab system reaches the end of its track section, it requests the permission to enter from that track section control that is responsible for the track section that is at this point the RailCab's next track section. In particular, we would also like to express that it need not necessarily be a track section control that the RailCab sends the request to—it could also be the control of a switch. Figure 3.9 illustrates how we would like the MSD `RequestEnterAtEndOfTrackSection` to apply to such a configuration of objects in a larger RailCab system. In particular, the figure shows a situation where a RailCab approaches a merging switch, and the next track section control is thus the control of a merging switch. This is a situation that may occur with different combinations of objects and we would like to specify one MSD for all of these combinations.

For this purpose, the concept of *symbolic lifelines* was introduced by Harel and Marelly [HM03]. Symbolic lifelines are lifelines that, instead of referring to one concrete object, refer to a class. Graphically, a symbolic lifeline can be recognized by a label that is not underlined. An MSD with only symbolic

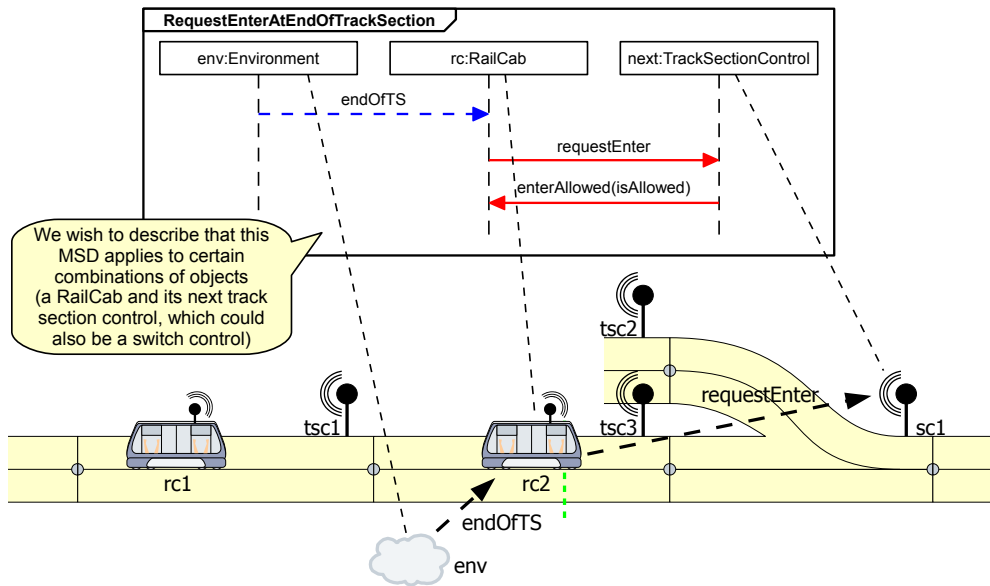


Figure 3.9: An example the `RequestEnterAtEndOfTrackSection` with symbolic lifelines in a dynamic system: which object does a lifeline represent?

lifelines is also called a *symbolic MSD*; an MSD with concrete lifelines is also called a *concrete MSD*.

Symbolic lifelines can be *bound* to objects in the object system that are instances of the class that the lifeline refers to. For a given object system, the semantics of a symbolic MSD is equivalent to a set of concrete MSDs where for each possible combination of bindings of the symbolic lifelines, there exists a concrete MSD with lifelines corresponding to this possible combination of bindings. In the following, this is called the *general semantics* of symbolic MSDs.

In some cases, we want to restrict the symbolic MSD to specify the behavior only for certain combinations of objects. Then, *binding expressions* can be added to the MSD in order to restrict the possible bindings for the lifelines. Binding expressions are Boolean expressions that refer to properties of objects that may be bound to the lifelines [HM03]. A symbolic MSD then only specifies the behavior for the combinations of objects where there exists a set of lifeline bindings where all binding expressions evaluate to true.

Due to the combinatorial explosion, mapping a symbolic MSD to an equivalent set of concrete MSDs is inefficient for the play-out of symbolic MSDs. Especially in large object systems where object properties change, or objects are added and removed from the system, maintaining an equivalent set of concrete MSDs is impractical. Therefore, the play-out algorithm interprets the symbolic MSDs differently. The resulting behavior, also called the *symbolic play-out semantics*, is not equivalent to the play-out of a corresponding set of concrete MSDs, also called the *general play-out semantics* in the following. There are small differences, which will be explained shortly.

The play-out semantics of symbolic MSDs is explained in the following. For this purpose, we first need to extend the concept of *event unification*. Next, we introduce *binding expressions*, which are restricted for the play-out algorithm to a certain form for an efficient operational interpretation. In order to express the relationships between objects, also the concept of *associations* is introduced.

According to Maoz [Mao09], we also consider the case where there are *generalization* relationships between the classes. Generalization is a directed relationship between two classes, where one class is the *general* class and the other is the *specific* class. The relationship means that an object of the more specific class is also an instance of a more general class. The specific class also inherits all the properties defined for the general class [UML09, Sect.7.3.20, pp. 71].

Event unification in the play-out of symbolic lifelines

On the occurrence of a message between two objects, the first message of an MSD can be unified with that message

1. if the operation of the message in the object system equals the operation of the diagram message and
2. if the class of the sending object is equal to or a specialization of the class represented by the sending lifeline.
3. if the class of the receiving object is equal to or a specialization of the class represented by the receiving lifeline.

If a message in the object system can be unified with the first event of an MSD, an active instance of that MSD is created. In this process, the sending and receiving lifelines of the first message are *bound* to the sending resp. receiving objects of the system message. Also the cut progresses beyond the first diagram message. The lifelines in the active MSD that are not the sending or receiving lifeline of the first message remain *unbound* unless, as explained shortly, there is a binding expression that determines a binding for the lifeline.

If a monitored or executed message is subsequently enabled in the MSD where the sending or receiving lifeline is unbound, there exist sophisticated concepts for how the play-out algorithm can best mimic the general play-out semantics of the symbolic MSDs. Harel and Marelly have elaborated these concepts for LSCs [MHK02, HM03], but they could be also applied to MSDs. These concepts involve creating duplicates of active MSDs/LSCs so that sequences of events among different objects that the lifelines can be bound to can be monitored or messages can be sent to multiple objects.

For simplicity, in the scope of this thesis, it is required that if a message is enabled, the sending and receiving lifelines must be bound. Then it is determined which message in the object system the diagram message can be unified with, or, if the diagram message is executed, which message the play-out algorithm shall send between which objects. This can be achieved with binding expressions and relationships that exist between the objects. These relationships are expressed on the class level via associations, which binding expressions can refer to.

Associations

In a class model, there can be associations between two classes [UML09, Sect. 7.3.3]. An association can be *unidirectional* or *bidirectional*. A unidirectional association is displayed as a solid arrow between two classes; a bidirectional association is displayed as a solid line between two classes. A unidirectional association has one *navigable end* at the target end of the line with a name and a *cardinality*, which can be *single-valued*, written as 0..1, or *multi-valued*, written as 0..*. A bidirectional association has two *navigable ends* on both ends of the association line. A navigable end means that instances of the opposite class can have a pointer (in the single-valued case) or have a list of pointers (in the multi-valued case) to instances of the class at that end of the association. A bidirectional association means that if one object has a pointer to another object according to a navigable end of an bidirectional association, this object also has a pointer to the pointing object according to the opposite navigable end of the same association.

These pointers or lists of pointers can be modified by operations defined in the classes. By convention, in the case of a single valued navigable end, the operation `set<navigable-end-name>(newValue:<class-on-navigable-end>)` can be used to set the pointers of an object. In the case of a multi-valued navigable end, the operation `set<navigable-end-name>(newValue:List<class-on-navigable-end>)` can be used to set the lists of pointers of an object. To add a pointer to a list or to remove a pointer from a list of a multi-valued navigable end, the operations `add<navigable-end-name>(element:<class-on-navigable-end-name>)` and `remove<navigable-end-name>(element:<class-on-navigable-end-name>)` can be used. As described in Sect. 3.1.11, messages referring to these operations can be used in MSDs to specify the changes of pointers in the object system. Furthermore, we allow diagram variables in MSDs that bind to objects or lists of objects.

A pointer to an object is also called a *link*. The pointing object is called the *linking* object; the object pointed to is also called the *linked object*. One navigable end of an association can also be a *composition*, displayed as a black diamond at the opposite end of the association line. A composition means that a linked object is “part of” the linking object. Within the scope of this thesis, it is defined that an object can only be linked by one link that corresponds to a navigable end that is a composition, i.e., a object can only be part of one object at a time.

Figure 3.10 shows parts of a RailCab object system and its class diagram. This part of an object system corresponds to the system graphically displayed above in Fig. 3.9. The class diagram has a bidirectional and one unidirectional association. The bidirectional association expresses that a track section control can have many registered RailCabs. The registered RailCabs have the track section control as their current track section control. The unidirectional association says that a track section control can have one next track section control. Furthermore, there is a generalization relationship from the class `MergingSwitchControl` to `TrackSectionControl`, displayed by an arrow with a hollow triangle arrowhead. This generalization means that instances of `MergingSwitchControl` are also in-

stances of `TrackSectionControl`. The object system shows a number of instances of these classes with a certain configuration of links. Links are displayed as arrows labeled with the name of the corresponding navigable end. (In order to build meaningful track systems, we would of course also need a branching switch control. But, these are not considered here to keep the examples small.)

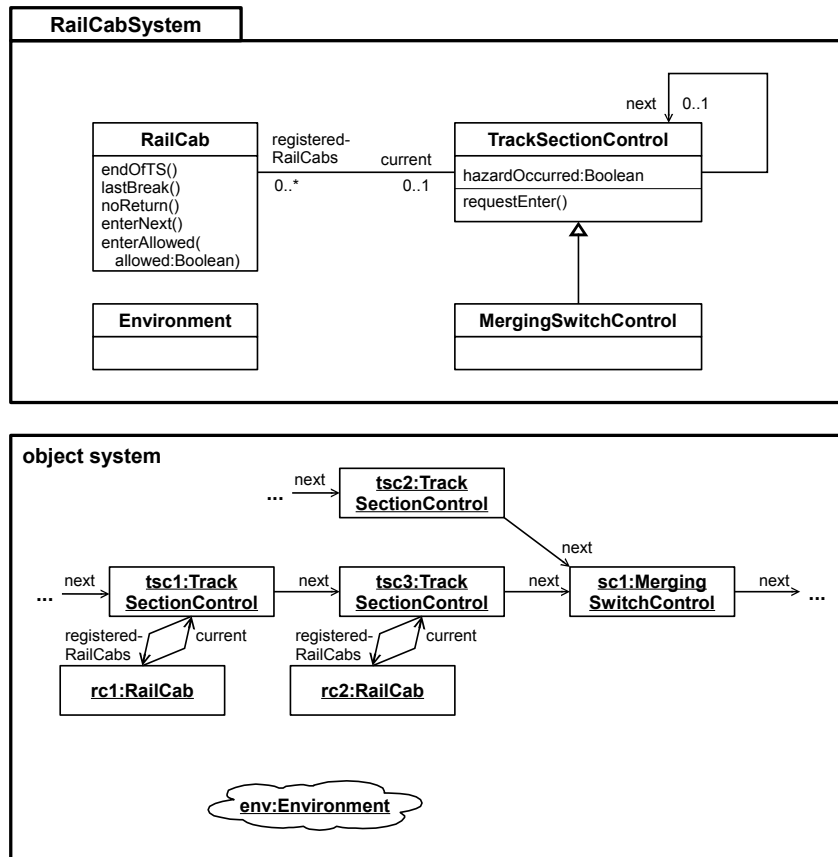


Figure 3.10: A class model of the RailCab system and a possible object system (an object diagram representation of the system shown in Fig. 3.9)

Binding expressions

If a message is enabled in an active MSD, we require that the sending and receiving lifeline is bound. To ensure this, *binding expressions* must be specified for all lifelines in a symbolic MSD that are not the sending and receiving lifeline of the first event. Remember that these lifelines will be bound during the unification of the first event. The binding expressions have the form `<lifeline-name> := <expr>`, where `<lifeline-name>` is the name of a lifeline, called *slot lifeline*, and `<expr>` is an OCL expression [OCL10], called the *value expression*, that evaluates to an object or a set of objects that are instances of the class that types the role referred to by the slot lifeline. The result of evaluating the value expression can also be null or an empty set, respectively. Again, diagram and

lifeline variables can be used in the value expressions. The lifeline variables in the case of symbolic lifelines are bound to the same object that the lifeline is bound to. It may, however, also be that a lifeline variable, just as the lifeline, is *unbound*. The value expression cannot be evaluated if a variable that appears in it is unbound. The value expressions are evaluated as soon as they can be evaluated. Then the binding of the lifeline takes place as follows.

The value expression evaluates to an object: If the value expression results to an object or a set containing a single object, the lifeline is bound to that object.

The value expression evaluates to null: If the value expression results to null or an empty set, this is a *violation* of the active MSD. This means that

- it is a safety violation if the current cut of the active MSD is hot.
- it is a cold violation if the current cut of the active MSD is cold, i.e. the active MSD will be discarded
- no active MSD will be created if the expression is evaluated right after unifying the first message of an MSD.

The value expression evaluates to a set of two or more objects: If the value expression results to a set of two or more objects, then copies of MSDs are created for each object so that each object is bound to the slot lifeline in one active MSD.

Figure 3.9 shows an example of the symbolic version of the MSD `RequestEnterAtEndOfTrackSection`, which contains a binding expression for the third lifeline. (The expression label is attached to the according lifeline for better readability, but the connector carries no additional semantics.) The expression says that the `next:TrackSectionControl` lifeline must be bound to the object which is the next track section control of the current track section control of the RailCab bound to the lifeline `rc:RailCab`. For the RailCab `rc2` in the object system illustrated in Fig. 3.9, this would be the section control `sc1`.

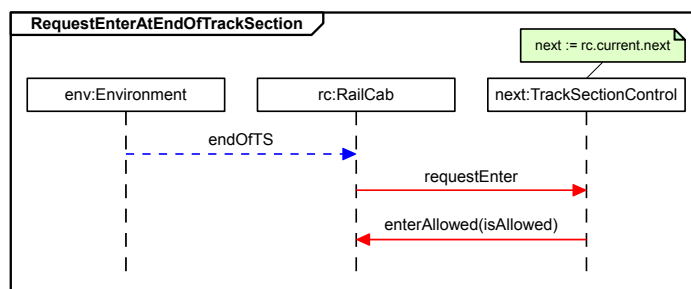


Figure 3.11: The MSD `RequestEnterAtEndOfTrackSection` with symbolic lifelines and a binding expression

If the event `endOfTS` occurs in the object system, first an active copy of the MSD is created with a binding for the lifelines `env:Environment` and `rc:RailCab` as explained above. Then the binding expression can be evaluated, which results in a binding of the lifeline `next:TrackSectionControl`. If a binding expression cannot be evaluated until a message attached to that lifeline is enabled, the behavior

is undefined. In SCENARIOTOOLS, a play-out algorithm is implemented, which reports this to the user as an exception.

The difference between the symbolic play-out and general play-out semantics

Due to the restriction above, that the sending and receiving lifelines of enabled messages must always be bound, the symbolic play-out semantics is very similar to the general play-out semantics. However, there is one detail where the semantics differ. Remember that it was defined that it is a safety violation if a hot message is enabled in an active MSD, but another event occurs in the object system that is unifiable with another message in the MSD that is not currently enabled. The problem is that, for practical reasons, we do not require that all lifelines must be bound immediately after the activation of the MSD. We can, therefore, sometimes not decide for every message in a symbolic MSD, whether an event that occurs in the object system is unifiable with that diagram message as long as its sending or receiving lifeline is unbound. All the cases where this cannot be decided are ignored. The symbolic play-out semantics may therefore produce executions where according to the general play-out semantics of symbolic MSDs safety violations would occur.

3.1.16 Forbidden messages

As explained above, if a hot message is enabled, this means that certain messages are not allowed to occur. Sometimes, however, we would like to specify that also other messages, which do not necessarily represent an event in the scenario, are forbidden to occur while an MSD is active.

Harel and Marelly introduced the concept of *forbidden messages* [HM03, Chap. 17] in order to describe events that must not occur during a scenario or events that may interrupt a scenario. Within this thesis, the convention is used that forbidden messages are messages that have the `forbidden` stereotype applied, indicated by the «forbidden» label attached to the message arrows, and that must only occur in a section at the end of an MSD that is separated from the rest of the diagram by a cold `false` condition that covers all lifelines of the MSD. This condition marks the actual end of the diagram; there is a cold violation once the condition is enabled, thus having the same effect as the termination of the MSD.

Forbidden messages have a temperature, but no execution kind. Hot forbidden messages represent events that are not allowed to occur while the MSD is active unless a corresponding message is currently enabled. Cold forbidden messages represent events that can interrupt active MSD by a cold violation unless a corresponding message is currently enabled.

Figure 3.12 shows an extension of the MSD `RequestEnterAtEndOfTrackSection` that now has one cold and one hot forbidden message. The meaning of this MSD is that, while the MSD is active, an occurrence of the message `cancelRequest` (sent between the objects represented by the lifelines `rc` and `next`) leads to a cold violation of the active MSD, which is therefore discarded.

It is even a cold violation if the current cut is hot. An occurrence of the event `enterNext` (between the objects represented by the according lifelines) while the MSD is active constitutes a safety violation, even if the current cut is cold.

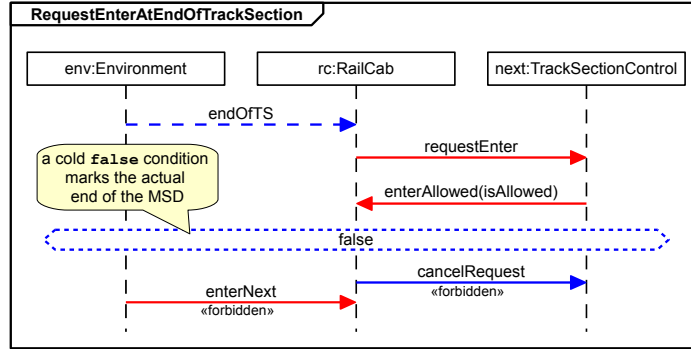


Figure 3.12: The MSD `RequestEnterAtEndOfTrackSection` with additional forbidden messages

3.2 Timed Game Automata and Uppaal TIGA

Because time is an important aspect for many kinds of systems, models for reactive systems were introduced in the past that consider time, for example *Timed Automata* [AD94]. Timed Automata are finite automata with a finite number of *clock variables* that can take positive real values and synchronously, monotonically, and continuously increase with time. The value of these clock variables can be reset on state transitions. Such a time model is called a *real-time* model or, synonymously, *dense-time* or *continuous-time* model. Real-time models have advantages over discrete-time models. For example is it not necessary to a priori decide on adequate time intervals for discretizing the model of a given system.

Alur et al. have shown that by restricting the expressions over clock values, Timed Automata can be described by a finite transition system [ACD93], which makes the automated checking of temporal properties feasible. A tool for modeling networks of Timed Automata and automatically checking their properties is the model checker UPPAAL [BLL⁺96]. A number of other analysis tools were implemented on the basis of the UPPAAL model checker, such as UPPAAL TIGA [CDF⁺05, BCD⁺06, BCD⁺07a], a tool for finding strategies in timed and untimed two-player games.

In this thesis, a technique for synthesizing controllers and for finding inconsistencies in MSD specifications was developed based on UPPAAL TIGA (see Chap. 4). In the following, this section gives a short introduction to the Timed Automata (TA) in UPPAAL and the Timed Game Automata (TGA) in UPPAAL TIGA. For a more comprehensive introduction to UPPAAL and UPPAAL TIGA, see the UPPAAL tutorial [BDL04] and the UPPAAL TIGA manual [BCD⁺07b]. A summary on the principles of the on-the-fly strategy synthesis algorithm implemented by UPPAAL TIGA is given in Sect. 3.2.3.

For the purpose of mapping MSD specifications to a TGA model (see Chap. 4), an EMF-based meta-model which reflects parts of the language concepts of UPPAAL resp. UPPAAL TIGA was created. This meta-model is presented in Appendix A.1.

3.2.1 Timed Automata in Uppaal

In the UPPAAL model checker, real-time systems can be modeled by a system of parallel Timed Automata. Figure 3.13 shows an example of a light switch modeled as a system of two parallel Timed Automata. The example is taken from the UPPAAL tutorial [BDL04].

Such a system is described by a number of *templates* of Timed Automata, *global declarations* of variables, clocks, channels and functions, as well as a *system definition*, which defines how the templates are instantiated to run as concurrent processes. The templates are specified graphically; the variables, clocks, channels and functions are specified in a C-like textual notation (see the UPPAAL [BDL04] tutorial for details on the textual notation). Figure 3.13 shows a system consisting of two template instances: `Lamp1` is an instance of template `Lamp` and `User1` is an instance of template `User`.

A template defines a Timed Automaton, including *local declarations* of variables and functions that are used in the automaton. For clarity, a template is called an *automaton template* in the following. The automaton in an automaton template consists of a set of *locations* and *edges*. Each automaton has one *initial location*, marked by the double-bordered circle. In a system, each instance of a template can be in one *current location* at a time. Locations and edges are deliberately not called states and transitions, since the *state* of a system is defined by the set of current locations of all template instances and the valuation of the variables (including the clock variables). A *transition* is a discrete change in the state of the system, which may be one edge firing or multiple, synchronized edges in different template instances firing at the same time, which possibly, in doing so, update the values of the variables and resetting clock values.

Each edge has a number of labels. Important here are the *guard*, *update*, and *synchronization* labels. Guard labels are side-effect free expressions over variables and clocks that evaluate to a Boolean value. Clock guards are restricted to the form $x_1 \bowtie expr$ or $x_1 - x_2 \bowtie expr$, where x_1, x_2 are clock variables, $expr$ is an integer expression, and \bowtie is a compare operator $\bowtie \in \{<, \leq, >, \geq, =\}$. Update expressions are a comma-separated list of assignments that assign values to variables. Clock variables may only be reset to positive integer values.

An edge is *enabled* in a template instance when the source location is the current location of the template instance and when its guard expression evaluates to true. Guards are evaluated before the updates are executed. Synchronization labels are used to synchronize edges via *channels*. The labels `press!` and `press?` in Fig. 3.13 are examples for such synchronization labels. In this case `press` is a *binary channel*, which means that, when the sending edge, labeled `press!`, fires, then a currently enabled receiving edge, labeled `press?`, must fire synchronously. If more than one receiving edge is enabled, one of these enabled edges is non-deterministically chosen for synchronization. If no receiving edge is enabled, the

sending edge cannot fire. (The two latter cases will, however, never occur in the lamp switch example.)

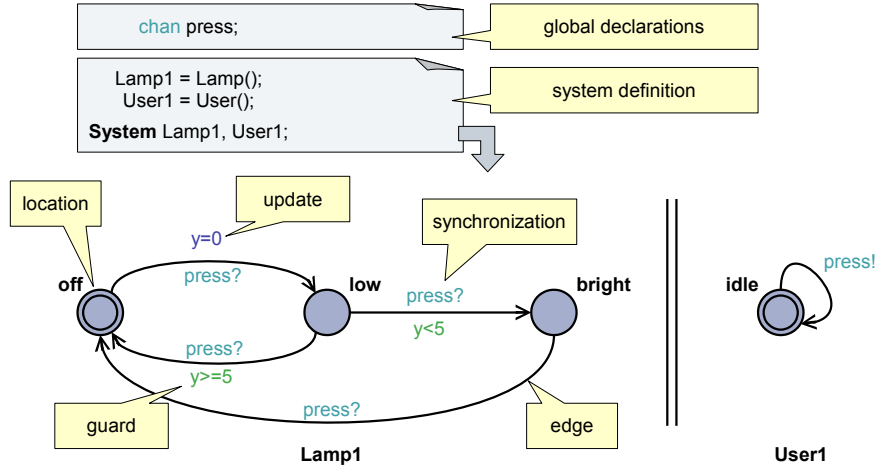


Figure 3.13: An example of a light switch modeled as a system of Timed Automata in UPPAAL

Channels can further be *broadcast channels*, which means that a sending edge must synchronize with an edge in each template instance where an according receiving edge is enabled. If there are multiple enabled receiving edges in a template instance (all going out of the current location), one is chosen non-deterministically. Also, in contrast to the binary channel, the sending edge may fire when no receiving edges are enabled.

In addition to the name label, locations may have an *invariant* and they may be *urgent* or *committed* locations. The location invariant is a side-effect free, Boolean expression, like the guard expression of an edge. The only restriction is that lower bounds on clocks are not allowed. The system can never be in a location when the invariant of the location evaluates to false. Therefore, the system must leave the location before the invariant is false and it must not enter a location when the invariant evaluates to false. This further has implications on the synchronization: for example an enabled edge sending over a broadcast channel cannot fire when an enabled receiving edge leads to a location where the invariant evaluates to false.

If a location is urgent, this means that no time is allowed to pass while the template instance is in this location. If a location is committed, no time is allowed to pass, just as for urgent locations. But yet more strictly, it is required that the next transition leaves a committed location. If the system is in multiple committed locations at once, only transitions can be taken which leave committed locations.

The specification language used in UPPAAL is a subset of CTL. UPPAAL can check predicates over locations, variables and clocks. Furthermore, UPPAAL can check properties $AG \varphi$ or $EG \varphi$. Here φ is a predicate and $AG \varphi$ means that on all paths (A) always (or “globally”, G) φ holds. $EG \varphi$ means that there exists (E) a path where always (G) φ holds. Also reachability properties

in the form of $EF \varphi$ can be checked. This property means that there exists a path (E) where eventually (or “finally”, F) φ holds. Finally, UPPAAL can check liveness properties of the form $AF \varphi$ or $\varphi \rightsquigarrow \psi$. $AF \varphi$ means that on all paths (A) eventually (F) φ holds. $\varphi \rightsquigarrow \psi$ (“ φ leads to ψ ”) is equivalent to $AG (\varphi \Rightarrow AF \psi)$, meaning that on all paths (A) it is always the case (G) that if φ holds, then on all paths (A) eventually (F) ψ holds. Path formulae (i.e. temporal operators) are not allowed to be nested. See the UPPAAL tutorial [BDL04] for details.

3.2.2 Timed Game Automata in Uppaal TIGA

UPPAAL TIGA [CDF⁺05, BCD⁺06, BCD⁺07a] is an extension of UPPAAL for synthesizing winning strategies in timed and untimed two-player games. UPPAAL TIGA extends the Timed Automata of UPPAAL to Timed Game Automata (TGA). In addition to the TA introduced above, edges in a TGA can be marked as *controllable* or *uncontrollable*. Transitions are controllable, if they involve only controllable edges, otherwise transitions are uncontrollable. Controllable transitions can be fired by the system, uncontrollable transitions can be fired by the environment. If controllable and uncontrollable transitions are enabled, the environment has priority over the system in firing the uncontrollable transitions. This implies that the system can only fire if the environment chooses to wait or has to wait, i.e., no uncontrollable transitions are enabled.

Given a network of TGA, UPPAAL TIGA can find a strategy for the system that satisfies a certain *winning condition* or it produces a counter-strategy, which shows how the environment can violate the winning condition. Winning conditions can be safety or reachability formulae, of the form $AG \varphi$ or $AF \varphi$, respectively. Furthermore, liveness properties can be specified of the form $AGAF \varphi$, which means that φ must be true infinitely often.

A strategy is a function that tells the system resp. environment to wait or to take certain transitions in a certain state. The strategy does not map all states potentially reachable in the TGA system, but only such which are reachable in a game controlled by the strategy, which means that there may be very small strategies for large systems. Strategies may be nondeterministic, and UPPAAL TIGA is able to compute *complete* strategies [CDF⁺05, BCD⁺07b], which means that for each state all transitions are computed by which the system or environment can win.

Since UPPAAL TIGA introduces no further syntactic constructs, an extensive example is omitted here. See the UPPAAL TIGA manual [BCD⁺07b] for details. The following section introduces the key idea of the strategy synthesis algorithm.

3.2.3 On-the-fly synthesis of game strategies

In UPPAAL TIGA, the winning strategies for the above-mentioned kinds of properties are determined by a backwards computation of *winning states*. The principle of the algorithm for solving reachability and safety games is briefly explained in the following. For details, see the original presentation of the algorithm by

Cassez et al. [CDF⁺05]. The algorithm for solving a game satisfying a liveness property has not yet been documented or published.

The basic game solved by synthesis algorithm in UPPAAL TIGA is a reachability game. In a reachability game, a strategy for the system must be found such that it can always reach a state that satisfies a certain property. These states are also called *goal states* in the following. Safety games are the complement of these strategies: if there is no strategy for the environment to always reach a forbidden state, i.e., goal state of the environment, the system wins the safety game.

The algorithm can be best explained when only considering the reachability game in an untimed model. The algorithm for finding winning strategies in untimed reachability games was first described by Liu and Smolka [LS98]: starting from the initial state, the algorithm explores transitions and states until it finds a state which fulfills the winning condition. This is the first winning state. Then, it reevaluates the predecessor states of the winning state: if all states reachable by uncontrollable transitions are also winning states and at least one controllable transition leads to a winning state, then the predecessor state is also a winning state. Finally, if the reevaluation can include the initial state into the set of winning states, the system can always win the reachability game. If that is not the case, there is no strategy for the system to win.

In the timed case, the algorithm was enhanced by Cassez et al. such that a symbolic representation of states is used, i.e., not the winning status of single states is computed backwards, but the winning status of symbolic sets of states. Furthermore, the backward computation of the winning states must also consider the predecessors of states in different *time regions*.

For the untimed case, the algorithm for finding winning strategies is shown to have a complexity that is linear with respect to the size of the state space (number of states plus number of transitions). In the timed case, the performance of the algorithm is not guaranteed to be linear, but in experiments the authors show that, by the help of a number of optimizations, the runtime of their algorithm is linear [CDF⁺05] to the size of the state space.

3.3 Triple Graph Grammars

The synthesis approach elaborated in the scope of this thesis (see Chap. 4) requires a complex model-to-model (M2M) transformation from a stereotyped UML sequence diagram model to a network of Timed Game Automata (TGA). This transformation was specified and implemented by a Triple Graph Grammar (TGG) [Sch94, KW07].

TGGs are a rule-based formalism for specifying mappings and transformations between models, and there exists a number of engines today which can execute model transformations based on TGGs, for example FUJABA [Wag06] or MOFLON [AKRS06]. A transformation engine called the TGG INTERPRETER for the transformation of EMF¹ models in ECLIPSE was developed in the scope

¹Eclipse Modeling Framework <http://www.eclipse.org/emf/>

of the author’s master’s thesis [Gre06]. This engine was extended by new concepts and features in the scope of this thesis (see Chap. 6 and Sect. 7.1). In the following, the basic principles of TGGs and TGG-based model transformations are explained and the advantages of TGGs are highlighted.

3.3.1 Why model transformation with TGGs?

TGGs have a number of benefits over other model transformation approaches. A TGG consists of rules that declaratively describe the relationship between model patterns of two or more kinds of models. This way, a mapping between models that are in some way structurally similar can be described conveniently. A transformation developer only has to specify these relationships and does not have to deal with “programming” the control structure of a translator program. A control structure is not vital for describing many model transformation problems. To the contrary, the control logic is likely to require additional design and maintenance efforts.

The Query/View/Transformation specification (QVT) by the OMG [QVT08] also defines two declarative transformation languages, QVT-Core and QVT-Relations. TGGs and the declarative QVT languages are very similar and share many concepts; the QVT-specification however still contains semantic ambiguities [Gre06, GK07, GK10]. Also, tool support for the declarative QVT languages today is still premature.

3.3.2 TGG structure and semantics

TGGs define sets of corresponding graphs. An element of this set is typically a triple consisting of two independent graphs that are linked via a third graph, called the *correspondence* graph. Because of this triple structure, such a graph is called a *triple-graph* in the following. These different graphs in a triple-graph are typed over different type graphs. The TGG rules are non-deleting graph production rules that describe how, based on a start graph or *axiom*, triple-graphs can be created. Triple-graphs that can be created by a TGG are called *valid* triple-graphs. The axiom is the smallest valid triple-graph.

Transferred to the “modeling world”, TGGs define sets of corresponding models where the independent models, in the following called *domain models*, are instances of different meta-models. The domain models are linked via a *correspondence model*, which is an instance of a correspondence meta-model. In the following explanations, the terms *model* and *meta-model* are used instead of graph and type graph. A triple-graph is called a *triple-model*. The TGG rules describe how, based on an axiom, triple-models can be created. Triple-models that can be created by a TGG are, accordingly, called *valid* triple-models.

TGGs cannot only be used for defining sets of valid triple-models. They can be operationalized for a number of *application scenarios*. For example, if a model of one domain is given, called the *source* domain in the following, TGGs can be operationalized to create a model of the opposite domain, called the *target* domain, as well as a correspondence model, such that the resulting models form a valid triple-model. This application scenario is called *forward transformation*.

Sometimes, a TGG can also be applied in the opposite direction, which is then called the *backward transformation*. If two domain models are given, a TGG can further be interpreted in order to create a correspondence model that links the two domain models, thereby determining whether the two domain models correspond to each other with respect to the TGG. There are a number of further application scenarios, but in the scope of this thesis, TGGs are used for forward transformations only. Details on that application scenario are explained shortly in Sect. 3.3.3.

The objects in the axiom are typically root objects that will contain the objects that will be added to the model by applying TGG rules (as explained shortly). Figure 3.14 shows an example of a TGG axiom from a TGG that maps an MSD specification to a system of Timed Game Automata (TGA). This mapping is called *MSD-to-TGA* in short, and is the basis of the synthesis technique that will be explained in Chap. 4. This axiom is a model where two domain model objects exist, a package and an NTA (abbreviation for “Network of Timed Automata”). In addition, there is a correspondence model object that has links to the package object and the NTA object. The package object and the NTA object are the root objects of an MSD specification and a TGA network, respectively. The two domains are called *UML* and *TGA*, the correspondence domain is called *UML2TGA*.

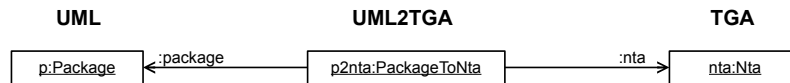


Figure 3.14: An example of an axiom.

Figure 3.15 shows an example TGG rule that is a simplified rule from the MSD-to-TGA transformation. A TGG rule is a graph grammar rule and consists of a left-hand side graph and a right-hand side graph. (Here they are displayed such that the left-hand side graph is at the top and the right-hand side graph is at the bottom.) Both sides consist of nodes and edges where a node is typed over a class in the meta-models of the according domain- or correspondence model; an edge is typed over a navigable end of an association in the meta-models of the according domain- or correspondence model².

A TGG rule can be applied to an existing triple-model if there is a valid *match* of the left-hand side pattern of the rule in the triple-model. There exists a match if a structure can be found in the triple-model that is isomorphic to the left-hand side pattern and the following conditions hold.

1. In a match, objects in the triple-model must be mapped to nodes such that the node’s type class equals the object’s meta-class. Such a mapping from a node to an object is called a *node binding*; in a node binding, the

²The TGG INTERPRETER relies on EMF, which defines a meta-meta-model according to the EMOF standard [MOF06]. In this meta-model, there exist no associations. Instead, classes can have *references* to other classes, which correspond to unidirectional associations. A bidirectional association can be represented by two references that are *opposites* to each other. Thus, an edge in a TGG rule of the TGG INTERPRETER is actually typed over a reference and not a navigable end of an association. This, however, has no other conceptual implications.

object is said to be *bound* to a node. Optionally, nodes can be marked as “*match subtype*”; then the object mapped to the node must be an instance of the node’s type class, i.e., it may also be an instance of a specialization of the node’s type class.

- For each edge in a match, a link from the object bound to the edge’s source node to the object bound to the edge’s target node must exist. Furthermore, the navigable end represented by the link must equal the edge’s type navigable end. A mapping from an edge in the rule to the source and target objects of a link as well as the navigable end of the link is called an *edge binding*; in an edge binding, the link is said to be *bound* to the edge in the rule.

When a match is found, the matched structure can be replaced by a structure as described by the right-hand side of the rule. But, because a TGG-rule is always a non-deleting graph grammar rule, when a match of the left-hand side pattern is found in the triple-model, the matched triple-model structure, instead of being replaced, can be extended by what is described by the additional nodes and edges on the right-hand side of the rule.

The rule in Fig. 3.15, for example, says that when a package and an NTA, connected via a particular correspondence object, is found in the model, a collaboration can be added to the package and two templates can be added to the NTA. (Templates or *automaton templates* define timed automata that can be instantiated in a system of timed automata, see Sect. 3.2.1.). In addition, two particular correspondence objects that link the newly added objects can be added to the existing correspondence model. One graph that this rule can be applied to is obviously the axiom.

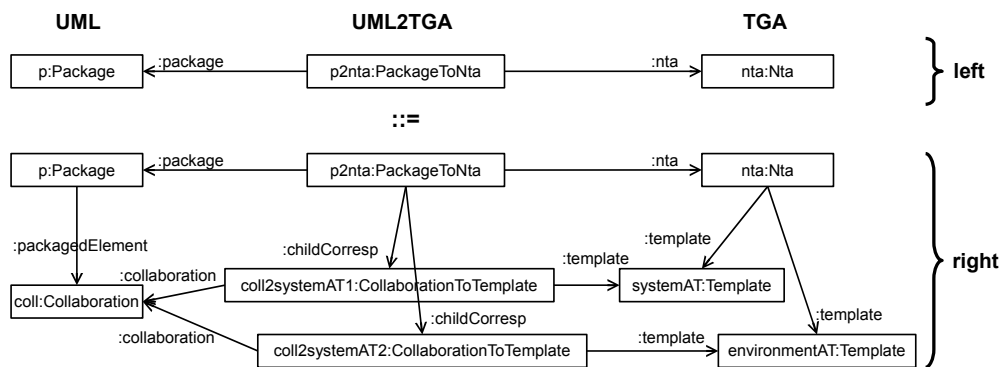


Figure 3.15: A TGG rule as a graph grammar production rule.

Because a TGG rule is a non-deleting graph grammar, another, short-hand notation can be used for displaying TGG rules. Such a short-hand notation for the TGG rule of Fig. 3.15 is shown in Fig. 3.16. The nodes and edges that occur on the left-hand side and the right-hand side of the TGG rule are called *context nodes* resp. *context edges* [GK10]. The nodes and edges that occur only on the right-hand side of the TGG rule are called *produced nodes* resp. *produced edges* [GK10]. Context nodes are displayed as a white box with a black border, context edges are represented by black arrows. The produced

nodes are displayed as green nodes with a “++” label. Similarly, the produced edges are shown as green arrows with a “++” label.

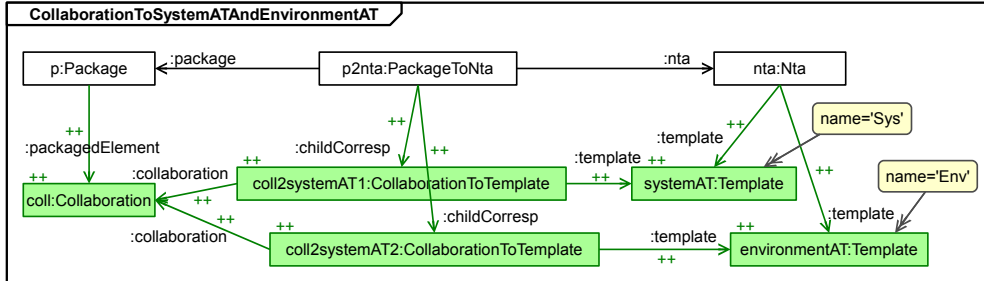


Figure 3.16: The short-hand notation of a TGG-rule with additional attribute value constraints.

Many extensions have been defined for TGGs. In addition to the TGG rule displayed in Fig. 3.15, the TGG rule displayed in Fig. 3.16 additionally contains *attribute value constraints*, displayed as yellow rounded rectangles. These constraints are attached to nodes and specify that the attributes of objects that are matched by the node or created according to the node must be equal to the given value. Here, one template object must be named ‘End’ and the other must be named ‘Sys’. Section 6.3 describes in more detail how attribute expressions can be formulated.

3.3.3 Forward transformation

A forward transformation works as follows. Let us assume that a source model is given and shall be transformed into a target model according to a given TGG. First, one instance of the axiom has to be created. That means that a target model pattern and a correspondence model pattern have to be created and connected to the appropriate objects in the source model such that there exists a model structure that is isomorphic to the axiom. This is illustrated in Fig. 3.17. The axiom is displayed as a rule, similar to a TGG rule as shown in Fig. 3.16. The objects and links in the model that make up the occurrence of the axiom are bound to the nodes resp. edges of the axiom graph. Note, however, that the axiom is not actually a rule and it cannot be “applied” multiple times.

Next, the TGG rules are applied in the following way. First, the context pattern and the source domain pattern are matched in the triple model such that context nodes and edges are matched only to already bound objects resp. links, and the produced source domain nodes and edges are matched only to unbound objects resp. links. If such a match can be found for a rule, the rule can be *applied* in the forward direction. This means that the produced target and correspondence patterns are created in the model and additional bindings are created for these newly matched resp. created objects. This forward rule application schema is illustrated in Fig. 3.18.

With a match of the axiom as illustrated in Fig. 3.17 the above rule `CollaborationToSystemATAndEnvironmentAT` can be applied in the forward transforma-

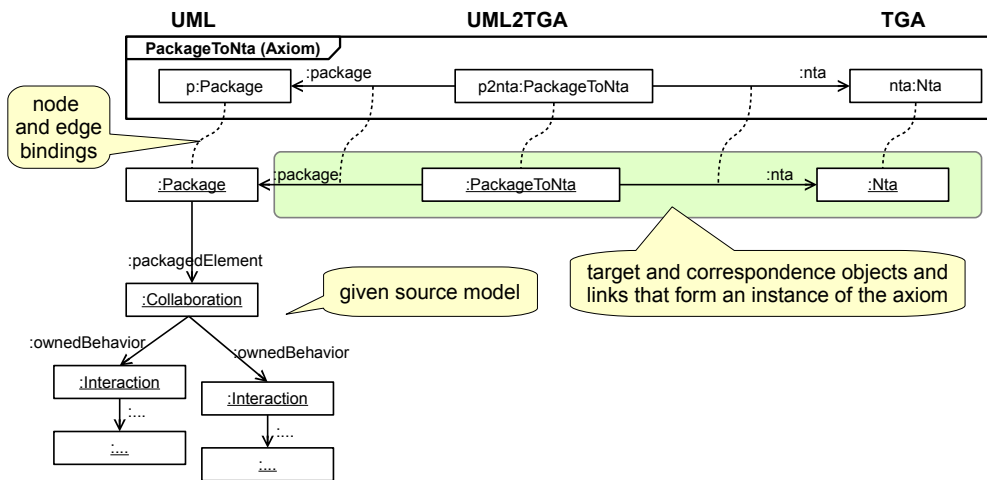


Figure 3.17: The creation of an instance of the axiom in the model.

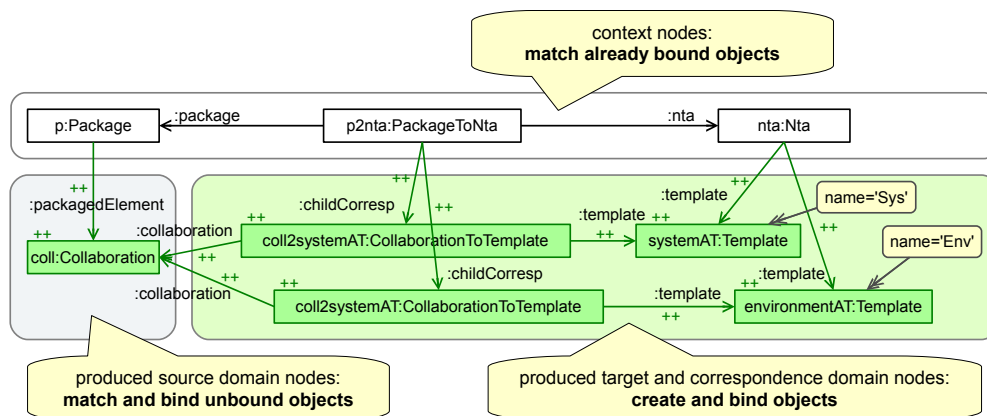


Figure 3.18: Interpretation of a TGG rule in a forward transformation scenario.

tion direction. The objects and links created in the correspondence resp. target domain as a result of the rule application are illustrated in Fig. 3.19. For the visual complexity, the bindings created for the nodes and edges in the rule to the objects and links in the model are omitted in this figure.

Following the above procedure, further TGG rules can be applied to transform the remaining source model. For example, there would be other rules for transforming the two interactions shown in Fig. 3.19 as well as further objects. The transformation terminates when no rule is applicable anymore. A transformation is successful when no unbound objects remain in the source model. In practice often only parts of the model have to be transformed. Then the transformation can be considered successful when all the elements in the regarded view are bound.

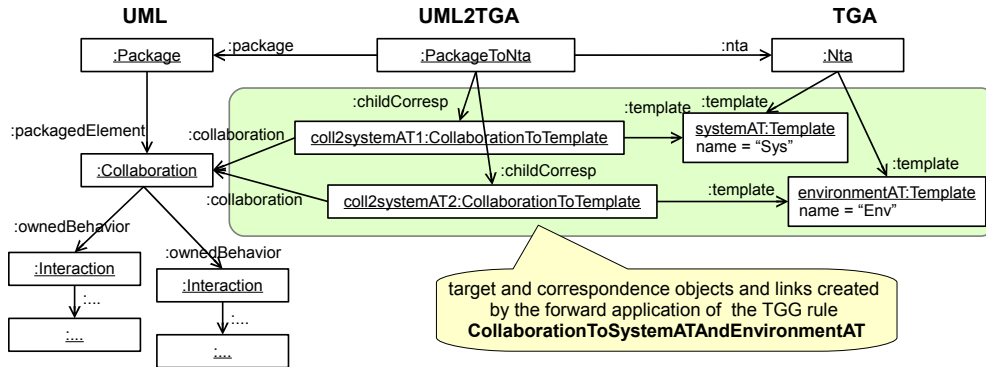


Figure 3.19: Result of the forward application of the TGG rule `CollaborationToSystemATAndEnvironmentAT`.

3.3.4 Further extensions of TGGs

Since the invention of TGGs, a number of extensions were proposed. Klar et al. for example propose a concept of *rule generalization*, which permits to reuse TGG rules in the definition of other, more specialized rules [KKS07]. Rule generalization may greatly reduce the number of rules that have to be specified. This concept is, with improvements, also employed in the TGG for mapping MSD specifications to a system of TGA as explained in Chap. 4. The improved generalization concept is described in more detail in Sect. 6.2.

Furthermore, the use of the Object Constraint Language (OCL) [OCL10], a powerful language for formulating queries in object-oriented models, has been proposed by Dang and Gogolla [DG08] to formulate attribute value constraints in TGGs. OCL attribute value constraints are also integrated in the TGG INTERPRETER. The details of this integration are explained in Sect. 6.3.

Another concept that was added in the scope of the author's master thesis [Gre06] and later more precisely defined in a paper [GK10], is the concept of *reusable nodes*. Reusable nodes are nodes that can be interpreted either as context nodes or as produced nodes. They may, for some mappings, greatly reduce the number of rules that have to be specified. A contribution of this thesis is a description of how the reusable nodes can be adequately interpreted operationally in a forward transformation scenario. See Sect. 6.5 for details.

Synthesis

This chapter presents a novel technique for synthesizing admissible controllers or admissible play-out strategies for untimed and timed MSD specifications that consist of a set of universal MSDs with concrete lifelines for a given, fixed object system. The novelty of this technique is that, in addition to MSDs that describe the requirements of the system, called *requirement MSDs*, environment assumptions given in the form of *assumption MSDs* can be considered by the synthesis. The synthesis can determine whether an MSD specification is consistent or consistently executable. If that is the case, it returns an admissible controller or an admissible strategy for the play-out algorithm, respectively.

Input for the synthesis is an MSD specification, including a description of an object system, in the form of a stereotyped UML model. This specification is mapped to a network of Timed Game Automata (TGA) in UPPAAL TIGA. The mapping is called *MSD-to-TGA* in short. With an adequate winning condition, UPPAAL TIGA can determine whether there exists an admissible controller for the system objects or an admissible strategy for the play-out algorithm. If there exists such a controller or strategy, it is returned by UPPAAL TIGA. If that is not the case, the specification is *inconsistent* or *not consistently executable*.

This chapter is structured as follows. Section 4.1 gives a brief overview of the approach. Input for the synthesis are UML-based MSD specifications given in a form as described in Sect. 4.2. Section 4.3 describes by an example how an untimed specification is mapped to a TGA network. The winning condition by which UPPAAL TIGA can synthesize an admissible play-out strategy for the specification is explained in Sect. 4.4. Section 4.5 presents variations and extensions to the MSD-to-TGA mapping that are necessary in a timed setting. Then, Sect. 4.6 presents a technique for the compositional synthesis of MSD specifications. This chapter first focuses on the problem of deciding consistent executability, but Sect. 4.7 then discusses changes to the MSD-to-TGA mapping by which other kinds of consistency can be decided. The mapping from an untimed or timed MSD specification to a TGA network is formalized and implemented by the TGG shown in Appendix B.

4.1 Overview

Figure 4.1 illustrates the principle of this approach. For each MSD specification, one *environment automaton* and one *system automaton* are created. Furthermore, each MSD and each assumption MSD is mapped to an *MSD automaton* or *assumption MSD automaton*, respectively. For clarity, we call an MSD that is not an assumption MSD a *requirement MSDs* in the following. A corresponding automaton is thus also called *requirement MSD automaton*.

The idea of these automata is that the environment and system automata can nondeterministically choose to “produce” environment resp. system events that appear in the specification. The edges in the system automaton are controllable, whereas the edges in the environment automaton are uncontrollable (indicated by dashed arrows). When events are “produced” in the environment or system automaton, certain edges in the MSD automata are synchronized via a broadcast channel. When fired, these edges update certain variables, this way representing the activation of MSDs, the progress of their cut, their termination, or cold violations and safety violations. An MSD automaton encodes the iterative semantics of an MSD, similar to the automaton shown in Fig. 3.3.

In addition to this encoding, a *winning condition* is specified. Intuitively, this condition expresses that, no matter which sequence of events the environment automaton produces, the system must

- avoid running into a safety violation of any requirement MSD and
- always eventually progress all active events and
- infinitely often listen for environment events

unless

- a safety violation occurs in an assumption MSD or
- an assumption MSD never progresses beyond an active event

If there is a way for the system to satisfy this condition, UPPAAL TIGA is able to generate a corresponding *strategy* for the system to do so. This strategy is a controller or an admissible play-out strategy that is a witness for the consistency resp. consistent executability of the MSD specification.

Note that this approach differs from a classical verification approach where typically the implementation of a system is given in the form of some transition system and is analyzed to satisfy a specification, typically formulated in temporal logic. Here the problem and its solution are different. The problem is that we search for an implementation to a given specification. The solution is that, first, we create a transition system of a system and its environment (the system and environment automata) that can virtually produce any possible sequence of events that appear in a specification. Second, we extend this transition system by the behavior represented by MSDs in the specification. Last, we specify a condition (the winning condition) that describes when a certain behavior of the system satisfies a given MSD specification. This condition is formulated in CTL, but it only describes what it means in general to satisfy an MSD specification, it does not itself specify any particular system requirements. In particular, this winning condition is the same for all MSD specifications (except for slight variations between the untimed and the timed setting).

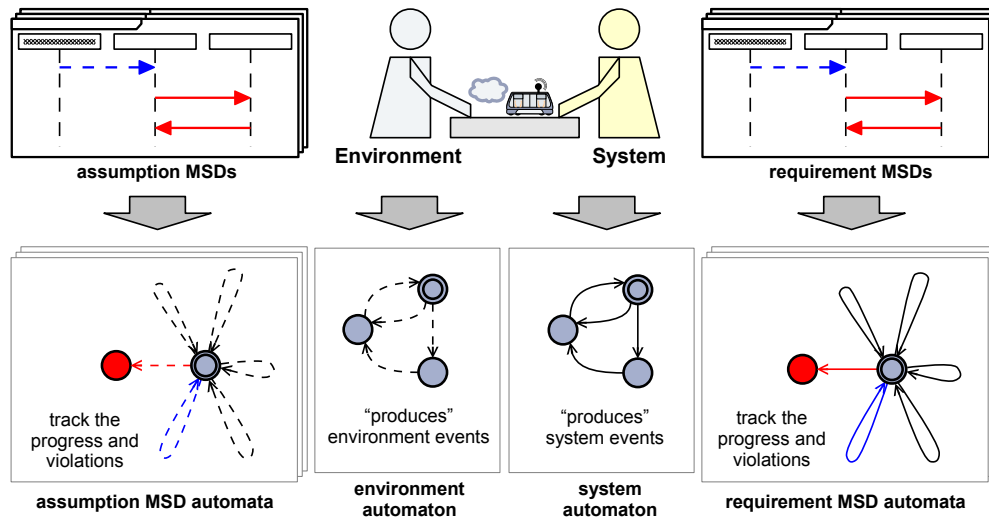


Figure 4.1: The principle of encoding an MSD specification as a system of Timed Game Automata

4.2 The MSD specification scheme

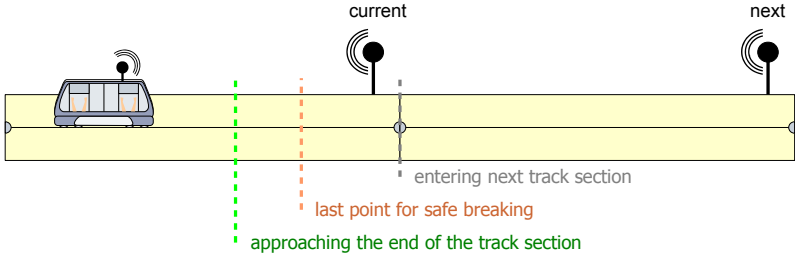
Input for the synthesis is an MSD specification, including the description of an object system, in the form of a stereotyped UML model. As an example, let us consider the MSD specification for the use case *Drive onto track section*, which is described informally in Tab. 4.1. This is a simplified version of the use case description shown in Fig. 2.3. For clarity, it is suggested to separate the description of the system requirements and the environment assumptions from each other in the informal use case descriptions.

4.2.1 Collaboration and class diagram

Figure 4.2 illustrates the UML-based MSD specification that captures the requirements and assumptions formulated above. The object system is described by a UML *collaboration diagram* [UML09]. A collaboration diagram is a special kind of UML composite structure diagram for describing instances that “collectively accomplish some desired functionality” [UML09, Sect. 9.3.3, p. 168]. The nodes in this diagram are called *roles*. Each role represents an object. Whether an object is a system object or an environment object, here again graphically indicated by a rectangle or cloud symbol, respectively, is modeled by a stereotype [UML09, Chap. 18] on the roles (see Fig. A.4).

The roles are typed over classes in a package. The top of the figure shows the class diagram of the package *DriveOntoTrackSection*. The classes have operations that specify which messages an object can receive. This class diagram is similar to the class diagram shown in Fig. 3.1. One difference, however, is that the parameter of the operation *enterAllowed* is omitted here. Parameterized messages are not yet supported by this synthesis technique. In this example,

Table 4.1: Use Case Drive onto track section

Use Case: Drive onto track section	Nr. 1
<p>Requirements: When the RailCab approaches the end of the track section, the RailCab must send a request to enter the next track section to the section control responsible for the next track section. Then the section control must reply, stating whether entering the track section is currently allowed or not. The reply must be received before the RailCab reaches the point of the last safe break.</p>	
<p>Environment assumptions: When the RailCab is notified that it approaches the end of the track section, it then passes the the point of the last safe break. At last, the RailCab enters the next track section.</p>	
<p>Sketch:</p> 	

the parameter on `enterAllowed` can be omitted, because the specification does not state whether it should allow the RailCab to enter the track section or not.

The lines between the roles are *connectors* that specify which objects interchange messages with each other. These connectors can be specified before modeling the MSDs to specify between which objects messages may be interchanged and, therefore, between which lifelines in the MSDs it is allowed to draw messages.

4.2.2 Requirement MSDs and assumption MSDs

Figure 4.2 shows the MSDs `RequestEnterAtEndOfTrackSection` and `ReplyBeforeLastSafeBreak`, which were already shown in Fig. 3.1. In the UML model, each lifeline of each MSD references a role in the collaboration. The message temperature and execution kind are specified by an extra message stereotype, as already proposed by Maoz and Harel [MH06] (see also Fig. A.5).

New here is the assumption MSD `LastBreakBeforeEnterNext`. *Assumption MSDs* specify how we assume the environment to behave. We assume that nothing will happen in the environment that will violate any assumption MSD. If the environment violates the assumptions, the system is not obliged to satisfy its requirements. The MSD `LastBreakBeforeEnterNext` specifies that after `endOfTS`, `lastBreak` must occur before `enterNext` occurs. For explanatory reasons, the

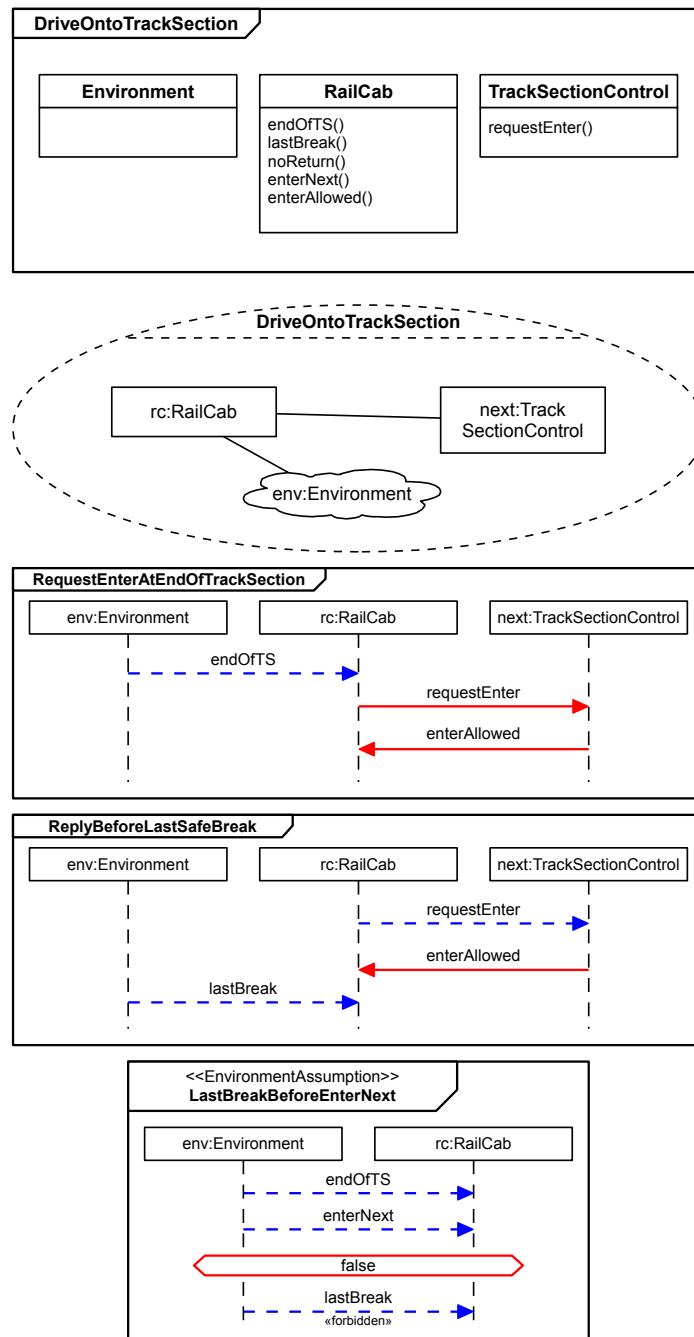


Figure 4.2: An untimed MSD specification, formalizing the requirements of the use case Drive onto track section

environment assumptions are modeled by an anti-scenario here: if `enterNext` follows `endOfTS` without `lastBreak` occurring in the meantime, the cut will be stuck in front of the hot false condition forever. This is therefore a liveness violation of the assumption MSD. Assumption MSDs are distinguished from requirement MSDs by the stereotype `EnvironmentAssumption`, which is graphically indicated in the MSD label.

The UML model structure and the used stereotypes are documented in more detail in the appendix, Sect. A.2.

4.3 Mapping untimed MSD specifications

An MSD specification given in the above form is mapped to a TGA network as follows. Section 4.3.1 explains the automaton templates for the system and environment automaton that are created for each MSD specification. Then Sect. 4.3.2 explains how the MSDs are each mapped to an automaton template.

4.3.1 The environment and system automata for untimed MSD specifications

The environment and system automata are shown in Fig. 4.3. The automata refer to a number of globally declared variables, constants, channels, and functions that are shown in Listing 4.1. Locally, the templates declare no further functions or variables.

The basic idea is that the environment automaton can choose which events occur in the environment, and the system automaton can choose which reaction the system performs. Both automata can “produce” events in two steps: first, the variable `event` is assigned with a value that represents the chosen event. Then the automaton emits over a broadcast channel, which may synchronize edges in the MSD automata. As will be explained shortly in Sect. 4.3.2, this leads to the activation of MSDs, the progress of cuts in active MSDs, the occurrence of cold violations or safety violations, or the termination of active MSDs.

Producing an environment event

Figure 4.3 shows the environment and system automata. Initially, the automata are in the locations `environmentInitial` and `systemInactive`. In this state, the environment may choose to take an uncontrollable edge to the location `produceEvent`. In this example, there are three edges, assigning either the value of the constant `env_rc_endOfTS`, `env_rc_lastBreak` or `env_rc_enterNext` to the variable `event`. These constants represent the environment events appearing in the specification. These are the messages `endOfTS`, `lastBreak` and `enterNext`, all sent from the object `env` to the object `rc`.

The location `produceEvent` is committed, which means that an edge leaving the location must be taken immediately, and before taking any other edge that is not leaving a committed location. In this case, the edge that leads to the location `handleHiddenEvent` must be taken. This edge emits over the broadcast channel events. Emitting over this broadcast channel may synchronize certain

edges in the MSD automata as will be explained shortly. Emitting over this broadcast channel also forces the system automaton to take the edge from the location `systemInactive` to the location `handleHiddenEvent`. (Because this system automaton edge is triggered by the uncontrollable edge in the environment automaton, and thus the system cannot control whether this edge is taken, this edge is also modeled as an uncontrollable edge).

In the locations `handleHiddenEvent`, the system and environment must progress all the currently active hidden events in the MSDs resp. assumption MSDs. Remember that hidden events are assignments and conditions that must be immediately progressed once they are enabled and before any other messages are sent in the system (see Sect. 3.1.12). This works as follows: Because the locations `handleHiddenEvent` in the environment and system automata are committed, edges leaving these locations must be taken immediately. The system automaton can take an edge to the location `systemActive`. This edge emits over the binary channel `exitHandleHiddenEvent`, and taking this edge therefore forces the environment automaton to synchronously move to the location `environmentInitial`. These edges can however only be taken if the guard expressions `not isHiddenEventEnabled(env)` and `not isHiddenEventEnabled(sys)` evaluate to true. The function `isHiddenEventEnabled(int player)` is a function that, if called with the parameter `sys`, returns true if there are hidden events enabled in the requirement MSDs. If the function is called with the parameter `env`, the function returns true if there are hidden events enabled in the assumption MSDs. Because of these guard expressions, the environment and system automata must take edges that loop in the `handleHiddenEvent` locations as long as there are hidden events enabled in any of the MSDs or assumption MSDs. These looping edges emit over the binary channels `hiddenEvent` resp. `hiddenAssumptionMSDEvent`, which synchronizes edges in the MSD automata resp. assumption MSD automata that progress the cut beyond enabled assignments or conditions. Eventually, all enabled hidden events will be processed and then the environment and system automata must progress to the locations `environmentInitial` and `systemActive`.

The location `produceEvent` has an invariant `not hotEnvViolation and envCutChanged`. This invariant keeps the environment from producing events if, first, a sequence of events has occurred that leads to a safety violation of an assumption MSD. In this case, as we will see later, the variable `hotEnvViolation` will be set to true in the violated assumption MSD automaton. This restriction is introduced for performance reasons, to reduce the state space described by the TGA system. Second, the invariant keeps the environment automaton from producing events that have no effect on the MSDs in the current state. The environment must not produce events that do not activate or terminate any MSDs, nor change the cut of any MSD. Allowing these events would unnecessarily blow-up the state space. The variable `envCutChanged` records whether a previous environment event has activated or terminated any MSD or changed the cut of any MSD. It is set to false when the environment automaton changes from the location `produceEvent` to the location `handleHiddenEvent`. As we explain later, the variable is immediately again set to true if this edge synchronizes edges in the MSD automata, thereby activating or terminating at least one re-

quirement or assumption MSD or changing the cut of at least one requirement or assumption MSD. In order to “win” against the system, i.e., to falsify the winning condition, the environment automaton will always try to avoid running into situations where `envCutChanged` remains false.

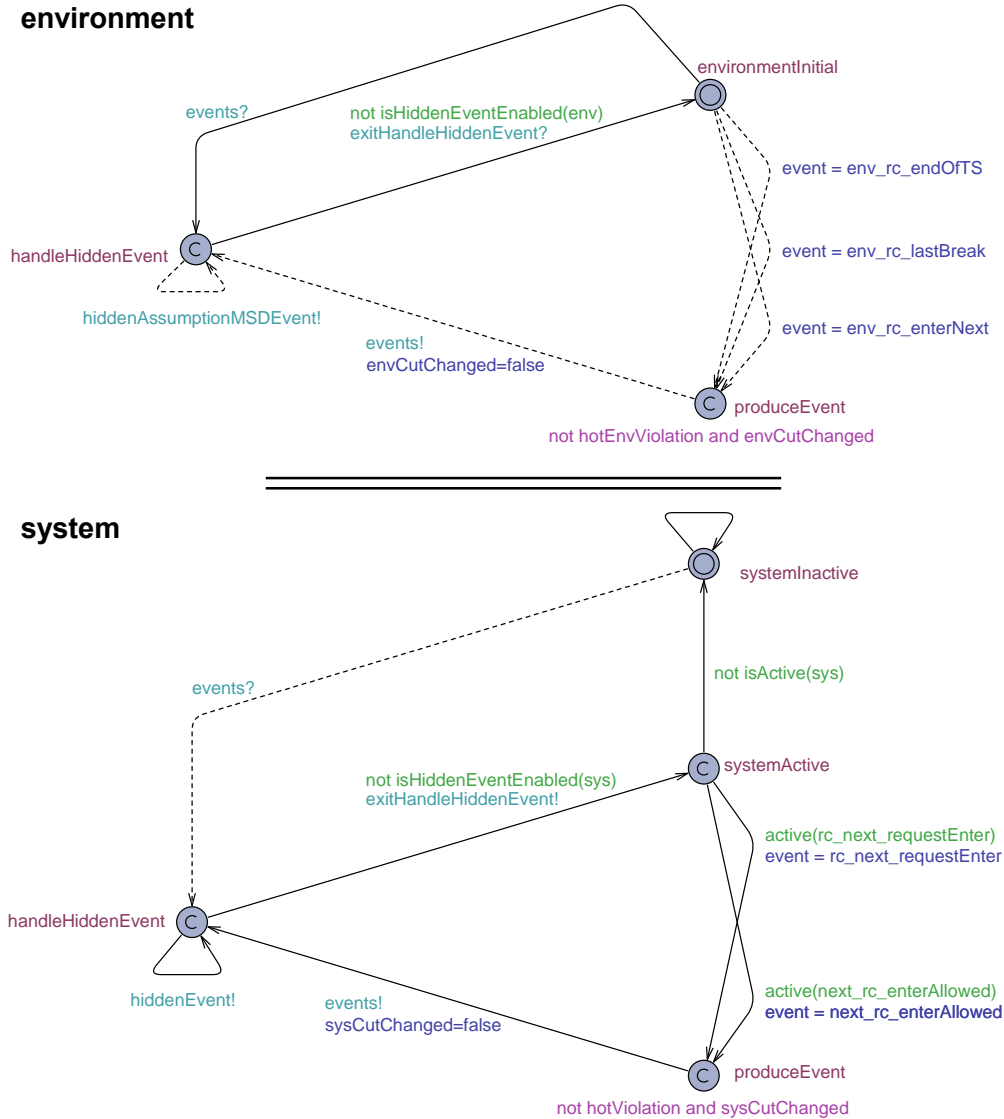


Figure 4.3: The environment and system automata for an untimed example MSD specification

Producing system events

When the environment and system automata are in the locations `environmentInitial` and `systemActive`, the system automaton must immediately take an edge leaving the committed location `systemActive`. This can be either the edge leading

back to the location `systemInactive`, or one of the edges leading to the location `produceEvent`. The edge back to the location `systemInactive` is guarded by an expression `not isActive(sys)`. The function `isActive(int player)` is a function that, if called with the parameter `sys`, returns true if there are active (i.e. enabled executed) messages in the requirement MSDs. If the guard expression evaluates to false, which means that there are active messages left to be executed by the system, one of the edges leading to the location `produceEvent` must be taken. These edges assign different constant values to the variable `event`. Each of these constants represents a message that is sent by a system object.

When the system automaton is in the location `produceEvent`, it must immediately take an edge to the location `handleHiddenEvent`. In doing so, like the environment automaton, the system automaton emits over the broadcast channel `events`, which may synchronize edges in the MSD automata. Emitting over the `events` channel also forces the environment automaton into the location `handleHiddenEvent`. Now, as explained above, both automata are again forced to process all hidden events before the environment and system automata can return to the locations `environmentInitial` and `systemActive`.

Note that the system is forced to immediately produce system events as long as there remain active events, and it can only produce events that are currently active in at least one active MSD. The environment can only choose to produce the next environment event once there are no active system events left and the system leaves the committed location `systemActive` to `systemInactive`. This behavior corresponds to the play-out semantics described in Sect. 3.1.8.

Furthermore, similar to the environment automaton template, the location `produceEvent` in the system automaton template has an invariant `not hotViolation and sysCutChanged`. This invariant keeps the system from producing events if a sequence of events has occurred that lead to a safety violation of a requirement MSD. Also, it keeps the system from producing events that do not activate or terminate any MSDs, nor change the cut of any MSD. These are again, as for the environment automaton template, measures to prevent an unnecessary blow up of the state space. (Note that, as explained so far, the system can only produce events if an according message is active in at least one requirement MSD. Consequently, producing a system event will in any case progress the cut of at least one MSD. Therefore, the variable `sysCutChanged` and the mechanism for keeping the system from producing events without any relevant effect is actually obsolete here. However, as will be discussed in Sect. 4.7.2, the guards on the edges to the location `produceEvent` must be removed if a certain kind of consistency is to be shown. Then the mechanism for keeping the system from producing events without any relevant effect becomes relevant again, and it is therefore included in the system automaton template for that particular case.)

In the system automaton, there is furthermore an edge that loops in the location `systemInactive`. This edge is important, because in the winning condition, which will be presented shortly in Sect. 4.4, we require that the system must be able to infinitely often enter that location. The system is able to do so infinitely often if it infinitely often reacts to environment events and eventually returns to the location `systemInactive` from the location `systemActive`. If, however, the

environment chooses not to produce any event at all, the system automaton can still take a transition where it loops in the location `systemInactive` and, this way, satisfy the winning condition.

Global declarations (part 1)

The declaration of variables, channels, constants and functions that the environment and system automata refer to are shown in Listing 4.1. The implementations of the functions depend on the MSDs in the specification. Therefore, the details of their implementation will be discussed in the following section, where it is explained how MSDs and messages, conditions and assignments are mapped to different parts in the TGA system. The language shown in the listings is the C-like language for declaring variables, clocks and functions in UPPAAL, see the UPPAAL tutorial [BDL04] for more information. The code comments explain the purpose of the created elements.

Listing 4.1: Global declarations (Part I)

```

/* Broadcast channel that synchronizes the edges in the MSD
   automata when 'producing' events in the system and
   environment automata. */
broadcast chan events;

/* This variable represents the event that the system
   or environment choose to 'produce'. */
int event = 0;

/* These variables represent whether a safety violation
   occurred in a requirement or assumption MSD,
   respectively. */
bool hotViolation = false;
bool hotEnvViolation = false;

/* These variables are set to false if an event
   produced by the environment or system does not
   change the cut of any MSD. */
bool envCutChanged = true;
bool sysCutChanged = true;

/* Binary channels that synchronize an edge in the system
   or environment automaton with an edge in one requirement
   or assumption MSD automaton for progressing the MSD's cut
   beyond enabled conditions or assignments, or for resetting
   the MSD's cut when it terminated. */
chan hiddenEvent;
chan hiddenAssumptionMSDEvent;

/* Binary channel that synchronizes the edges in the
   environment and system automata for synchronously leaving
   the 'handleHiddenEvent' locations. */
chan exitHandleHiddenEvent;

/* Constants defined for better readability;
   they are used in several functions. */

```

```

const int env = 0;
const int sys = 1;

/* These constants represent the different environment or system
   message events that appear in the specification. */
const int env_rc_endOfTS = 2;
const int rc_next_requestEnter = 3;
const int env_rc_lastBreak = 4;
const int env_rc_enterNext = 5;
const int next_rc_enterAllowed = 6;

/* returns whether there is any hidden event currently enabled
   in a requirement MSD (player = sys) or assumption MSD
   (player = env) */
bool isHiddenEventEnabled(int player){...}

/* returns whether there is any executed message or any
   hidden event currently enabled in a requirement MSD
   (player = sys) or assumption MSD (player = env) */
bool isActive(int player){...}

/* returns whether there is any executed message enabled in
   any of the MSDs that represents the given diagram event */
bool active(int ev){...}

/* additional global functions will be created for each
   MSD in the specification.*/
...

```

4.3.2 Mapping the MSDs to TGA

For each MSD in the specification, an MSD automaton template will be created. This section explains this mapping by the example of the MSD automaton template that is created for the MSD `RequestEnterAtEndOfTrackSection`, as shown in the above example specification (see Fig. 4.2). The automaton for this MSD is shown in Fig. 4.4. Listing 4.3 shows functions declared locally for that automaton template and Listing 4.4 shows additional global functions that are created for this MSD, and it shows fragments of the function bodies of the global functions that were introduced above in Listing 4.1.

Global declarations (part 2)

The basic idea of an MSD automaton is that it represents the progress of the cut of the MSD according to the iterative MSD semantics (see Sect. 3.1.6) upon the occurrence of environment or system events. It furthermore resets the cut when cold violations occur or the active MSD terminates, i.e., the cut reaches the end of the MSD. The cut is represented by a number of globally declared integer variables which represent the current location of the cut per lifeline. These variables are called *lifeline variables* and there is one lifeline variable created per lifeline and per MSD in the specification. These lifeline variables are named `<MSD-name>_<instance-name>`. Listing 4.2 shows the globally declared lifeline variables for the MSD `RequestEnterAtEndOfTrackSection`.

Listing 4.2: The globally declared lifeline variables

```

int RequestEnterAtEndOfTrackSection_env = 0;
int RequestEnterAtEndOfTrackSection_rc = 0;
int RequestEnterAtEndOfTrackSection_next = 0;

```

The MSD automaton

The MSD automaton consists of two locations. One location is the initial location, in which a number of edges loop. The second location is a location that represents the occurrence of a hot violation in the MSD. Except for the one edge that leads to the latter location, all edges loop in the initial location. The edges are all synchronized with edges in the environment or system automaton via the channels `events` or `hiddenEvent`. These edges represent different messages and hidden events in the MSD.

First, in each MSD automaton template, one edge is created that corresponds to the first event of the MSD (also called the *minimal event*). This edge is enabled in the initial cut, which means that all lifelines are at their initial position, (0,0,0) in this case. Additionally, the edge is enabled when the current event produced by the environment or system automaton is the first event in the MSD. The edge is synchronized by the broadcast channel `events`, and when triggered, the update expression increases all the lifeline variables to 1.

Second, each MSD automaton has edges for all the other messages in the MSD. For each message that is not the first, there is an edge with a guard that also checks that the currently produced event corresponds to this message and whether the lifeline variables are in a configuration which enables the sending and receiving of the message in the MSD. Instead of determining that directly in the guard expression, this is implemented in the Boolean function `enabled(int ev)`, which is created locally for each MSD template (see Listing 4.3). These edges are also triggered by the broadcast channel `events`. The update expression increments the lifeline variables of the sending and receiving lifeline. Figure 4.4 shows the edges created for the messages `requestEnter` and `enterAllowed`.

Third, each MSD automaton has an edge for handling the termination of the active MSD. Terminating the active MSD is a hidden event, and therefore the edge is synchronized via the binary channel `hiddenEvent`. The edge is enabled when the lifeline configuration has reached its maximal value, (1,3,3) in this case, and it updates the lifeline variables to the initial configuration (0,0,0).

Fourth, two edges are created in an MSD automaton which handle cold violations. One of these two edges is responsible for handling the particular case where the cold violation is caused by the occurrence of the diagram's first event. At the occurrence of a cold violation, the active copy of the diagram is discarded, which is expressed here by resetting the lifeline variables to their initial configuration. When the cold violation is caused by the first event, the active copy of the diagram is discarded, but a new active copy is created at the same time. This is expressed here by resetting all the lifeline variables of the violated MSD to 1. (This corresponds to the behavior of the Büchi automaton described in Sect. 3.1.6.) This behavior is realized by these two edges as follows.

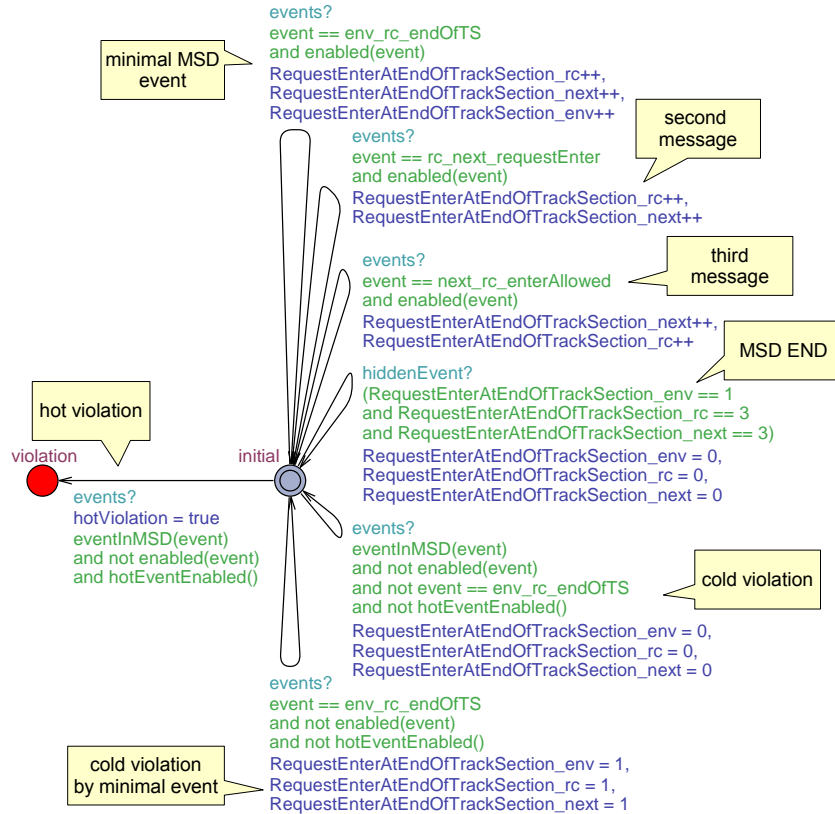


Figure 4.4: The TGA for the MSD RequestEnterAtEndOfTrackSection

The edge for the general-case cold violation is enabled if the current event is an event in the MSD that is not enabled. The former is determined by the Boolean function `eventInMSD(int event)` (see Listing 4.3). The latter is determined by the aforementioned Boolean function `enabled(int ev)`. Also, the current event must not be the first event of the MSD (`not event==env_rc_endOfTS`) and that no hot events are enabled in the current cut, because this would constitute a hot violation. Whether a hot event is enabled in the current cut is determined by the Boolean function `hotEventEnabled()` (see Listing 4.3).

The edge for handling a cold violation by a first event is only enabled if the current event is the first event of the MSD (`event==env_rc_endOfTS`). Furthermore, the event must not be currently enabled. The first event is enabled in the initial cut, in the configuration (0,0,0), but if the same message occurs multiple times in the MSD, it may also be enabled in other configurations.

Not shown in Fig. 4.4 is that all edges that change the values of the lifeline variables, i.e., all edges that loop in the initial location, set the variables `env-CutChanged` and `sysCutChanged` to true, in order to express that an event that was produced by the environment or system template automaton has changed the cut of at least one MSD (see in Sect. 4.3.1 that this is part of an optimization mechanism that keeps the system and environment from producing events that do not change the cut of any active MSD).

In the following, the implementation of the function declarations in Listing 4.3 and 4.4 are explained in more detail.

Declarations in the MSD automaton template

Listing 4.3 shows three Boolean functions created locally for each MSD automaton template. The first function `enabled(int ev)` consists of a return statement that is a disjunction of expressions which encode for each visible event (message) whether it is enabled in the current cut (i.e., in the current configuration of the MSD's lifeline variables). The function `hotEventEnabled()` returns true if the current cut is hot. Its return statement is a disjunction of expressions over the lifeline variables, which encode for each hot message and condition in the MSD whether it is enabled. The function `eventInMSD(int event)` returns true if the event encoded in the integer parameter `ev` is an event in the MSD. Its return statement consists of a disjunction of expressions which check for each visible event the equality of the parameter variable `ev` and the integer constant that is corresponding to the visible event in the MSD.

Listing 4.3: The functions declared in the automaton template for the MSD `RequestEnterAtEndOfTrackSection`

```

/* returns whether the event ev is enabled in the
   current cut of the MSD */
bool enabled(int ev){
    return((ev == env_rc_endOfTS)
           and (RequestEnterAtEndOfTrackSection_env == 0
                and RequestEnterAtEndOfTrackSection_rc == 0
                and RequestEnterAtEndOfTrackSection_next == 0)
           or (ev == rc_next_requestEnter)
           and (RequestEnterAtEndOfTrackSection_rc == 1
                and RequestEnterAtEndOfTrackSection_next == 1)
           or (ev == next_rc_enterAllowed)
           and (RequestEnterAtEndOfTrackSection_rc == 2
                and RequestEnterAtEndOfTrackSection_next == 2)
           );
}

/* returns whether any hot event is enabled in the
   current cut of the MSD */
bool hotEventEnabled(){
    return ((false
            or (RequestEnterAtEndOfTrackSection_rc == 1
                and RequestEnterAtEndOfTrackSection_next == 1)
            or (RequestEnterAtEndOfTrackSection_rc == 2
                and RequestEnterAtEndOfTrackSection_next == 2)));
}

/* returns whether the event ev is in the MSD */
bool eventInMSD(int ev){
    return (ev == env_rc_endOfTS
           or ev == rc_next_requestEnter
           or ev == next_rc_enterAllowed);
}

```

Global declarations (part 3)

Listing 4.4 shows globally declared Boolean functions, completing the declarations shown in Listing 4.1 and 4.2. First, there are the two functions of the form

- (1) `<MSD-name>_active(int ev)`, and
- (2) `<MSD-name>_hiddenEventEnabled()`

which are created for each MSD in the specification. Additionally, there are the Boolean functions

- (3) `active(int ev)`,
- (4) `isHiddenEventEnabled(int player)`, and
- (5) `isActive(int player)`

which are created once per MSD specification. The declarations of the latter functions were already shown in Listing 4.1. In the following it is explained how the bodies of these functions are composed of certain fragments that are created for each MSD in the specification.

Let us discuss the implementation principle of the function kinds (1) and (2) first. The function `RequestEnterAtEndOfTrackSection_active(int ev)` is an example of functions of kind (1). These functions consist of a return statement that is a disjunction of expressions that are created for each executed visible event and which encode whether the event is active in the current cut. The functions of kind (2) have the purpose to determine whether there are hidden events enabled in the current cut of the corresponding MSD. These functions consist of a return statement which is a disjunction of expressions to determine for each hidden event whether it is enabled in the current cut. Since the MSD `RequestEnterAtEndOfTrackSection` has only one hidden event, namely the event of its termination, the function `RequestEnterAtEndOfTrackSection_hiddenEventEnabled()` only returns whether the lifeline variables of the MSD have reached their maximal configuration (1,3,3).

The function `active(int ev)` determines whether the event encoded by the integer parameter is active in any of the MSDs. It implements a return statement which is the disjunction of calls to the functions of kind (1) explained above. Since only the MSD `RequestEnterAtEndOfTrackSection` is considered for now, just a call to the function `RequestEnterAtEndOfTrackSection_active(int ev)` is shown here. The dots state that further function calls to functions that correspond to the other MSDs in the specification will be added to the disjunction.

The function `isHiddenEventEnabled(int player)` checks whether there is any hidden event enabled in the assumption or requirement MSDs. It does so by a return statement that is a disjunction of calls to all the functions of the kind (2), in conjunction with either the expression `player == env` if the MSD is an assumption MSD, or in conjunction with the expression `player == sys` if the MSD is a requirement MSD. Again, since there is yet only one regular MSD regarded here, the disjunction of the return statement just contains a single conjunction statement. The dots state that further conjunction statements, corresponding to the other MSDs, will be added to the disjunction.

The function `isActive(int player)` determines whether there is any event currently active in any of the assumption or requirement MSDs. It does so by a return statement that is a disjunction of statements that are a conjunction of a call to the function `active(int ev)` and the expression `player == env` if the event is an environment event or the expression `player == sys` if the event is a system event. It is required that there are no executed environment messages in requirement MSDs and that there are no executed system messages in assumption MSDs. Additionally, the environment resp. system is considered active if there are hidden events enabled in any assumption MSD resp. requirement MSD. Therefore, a call to the function `isHiddenEventEnabled(int player)` with the parameter `player` is added to the disjunction.

Listing 4.4: Global declarations (Part II) – additions made for the MSD `RequestEnterAtEndOfTrackSection`

```

/* returns whether the event ev is active in the
   current cut of MSD RequestEnterAtEndOfTrackSection */
bool RequestEnterAtEndOfTrackSection_active(int ev){
    return ( false
            or (ev == rc_next_requestEnter
                and RequestEnterAtEndOfTrackSection_rc == 1
                and RequestEnterAtEndOfTrackSection_next == 1)
            or (ev == next_rc_enterAllowed
                and RequestEnterAtEndOfTrackSection_next == 2
                and RequestEnterAtEndOfTrackSection_rc == 2)
            );
}

...

/* returns whether any hidden event is enabled in the
   current cut of MSD RequestEnterAtEndOfTrackSection */
bool RequestEnterAtEndOfTrackSection_hiddenEventEnabled(){
    return (RequestEnterAtEndOfTrackSection_env == 1
            and RequestEnterAtEndOfTrackSection_rc == 3
            and RequestEnterAtEndOfTrackSection_next == 3);
}

...

/* returns whether the event ev is active in the
   current cut of any MSD */
bool active(int ev){
    return RequestEnterAtEndOfTrackSection_active(ev)
           or ...;
}

/* returns whether a hidden event is enabled in the
   current cut of any requirement MSD (player = sys) or
   any assumption MSD (player = env) */
bool isHiddenEventEnabled(int player){
    return ((player == sys and
            RequestEnterAtEndOfTrackSection_isHiddenEventEnabled())
           or ...);
}

```

```

/* returns whether any system event is active or any hidden
   event is enabled in the current cut of any MSD */
bool isActive(int player){
    return (isHiddenEventEnabled(player))
        or (
            RequestEnterAtEndOfTrackSection_active(rc_next_requestEnter)
            and player == sys)
        or (
            RequestEnterAtEndOfTrackSection_active(next_rc_enterAllowed)
            and player == sys)
        or ...);
}

```

The above example only covers a small subset of the language constructs defined for LSCs and MSDs. Encodings for asynchronous messages, combined fragments (sub-diagrams for loops or if-then-else constructs) are to be worked out the future. The encoding scheme for assignments and conditions is explained in the following.

4.3.3 Encoding assignments and conditions

To illustrate the encoding of assignments and conditions Fig. 4.5 shows an extended version of the MSD `RequestEnterAtEndOfTrackSection`. Here an assignment and a cold condition are added for the purpose of illustrating the encoding of these constructs. Note that these constructs can be applied in more meaningful ways once the synthesis is extended to consider object properties and parameterized messages. For now the main application for conditions is the terminal condition in anti-scenarios. Also, the principle of assignments and conditions is extended to clock resets and time conditions that can be used to formulate minimal and maximal delays between events. This is explained in Sect. 4.5.

In the MSD shown in Fig. 4.5, a diagram variable `i` is initialized with the value 0 after the message `endOfTS` was sent. Then the message `requestEnter` must be sent and the value of `i` is checked to be equal to 3. Since `i` was not changed since its initialization (remember that diagram variables are only visible within the scope of the active MSD, see Sect. 3.1.12), the condition will evaluate to false. Because it is a cold condition, this leads to a cold violation of the diagram.

Figure 4.6 shows the automaton created for the MSD in Fig. 4.5. An additional edge is created for the assignment and two additional edges are created for the cold condition. They are highlighted by labels and they are explained in the following. (Again not shown here is that all edges that loop in the initial location set the variables `envCutChanged` and `envCutChanged` to true.)

Assignments

Assignments are represented in the MSD automaton by an edge similar to the edge for the MSD END event as shown in Fig. 4.4. The guard of the assignment edge ensures that the current cut is immediately prior to the assignment on all lifelines that the assignment covers. The update label increases the lifeline

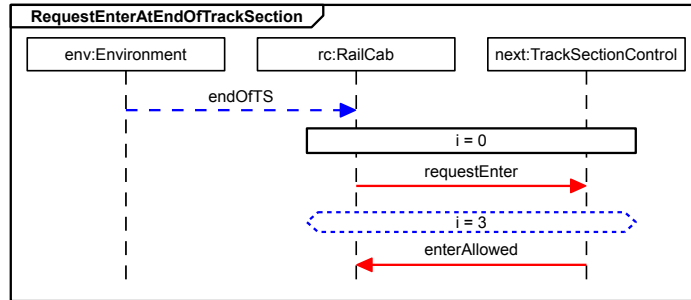


Figure 4.5: The MSD RequestEnterAtEndOfTrackSection extended by an assignment and a condition.

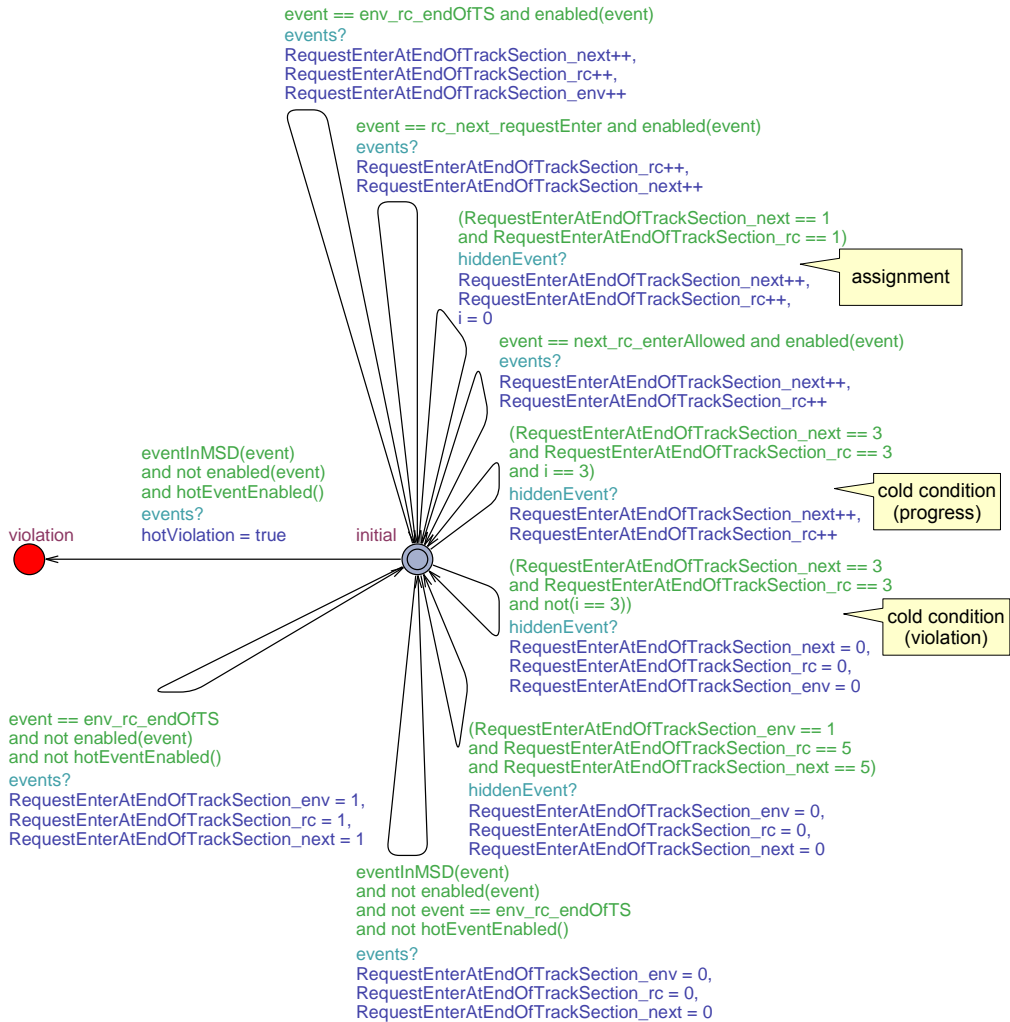


Figure 4.6: The TGA for the MSD RequestEnterAtEndOfTrackSection extended by an assignment and a condition

variables of the lifelines that the assignment covers. The assignment expression, here $i = 0$, is also part of the edge's update label. Assignment expressions are restricted to the form $\langle \text{var} \rangle = \langle \text{expr} \rangle$, where $\langle \text{var} \rangle$ is a integer variable and $\langle \text{expr} \rangle$ is a valid UPPAAL expression which evaluates to an integer value. Furthermore, $\langle \text{var} \rangle$ is mapped to an integer variable declaration in the declarations of the respective MSD template automaton.

Since the assignment is a hidden event, the edge is synchronized via the channel `hiddenEvent`. Also the global function `RequestEnterAtEndOfTrackSection_isHiddenEventEnabled()` is extended to return true when the assignment is enabled (see Listing 4.5).

Cold Conditions

As explained in Sect. 3.1.12, cold conditions are evaluated immediately. The active MSD progresses when the condition expression evaluates to true, and there is a cold violation of the active MSD (the active MSD is discarded) when the condition expression evaluates to false. This behavior is realized in the MSD automaton by two edges that are created per cold condition: one, called the *progressing edge*, which is enabled when the condition is enabled and the expression evaluates to true, and another, called the *violating edge*, which is enabled when the condition is enabled and the evaluation expression evaluates to false. The latter edge updates all lifeline variables to 0. Figure 4.6 shows the two edges.

Both edges are synchronized via the channel `hiddenEvent`. Also the global function `RequestEnterAtEndOfTrackSection_isHiddenEventEnabled()` is extended to return true when the condition is enabled (see Listing 4.5). Thus, either the progressing edge or the violating edge will be triggered by the system automaton immediately when the condition is enabled.

Listing 4.5: Global declarations (Part II) for the extended version of the MSD `RequestEnterAtEndOfTrackSection`

```

/* returns whether any hidden event is enabled in the
   current cut of MSD RequestEnterAtEndOfTrackSection */
bool RequestEnterAtEndOfTrackSection_isHiddenEventEnabled() {
    return (
        (RequestEnterAtEndOfTrackSection_env == 1
         and RequestEnterAtEndOfTrackSection_rc == 5
         and RequestEnterAtEndOfTrackSection_next == 5)
        or RequestEnterAtEndOfTrackSection_next == 1
         and RequestEnterAtEndOfTrackSection_rc == 1
        or RequestEnterAtEndOfTrackSection_next == 3
         and RequestEnterAtEndOfTrackSection_rc == 3
    );
}

```

Hot Conditions

When the cut reaches the hot condition, but the condition expression evaluates to false, this means that the MSD cannot progress beyond this condition until

it becomes true. It is a liveness violation only when that is never the case. Because this approach does not yet support object properties, hot conditions can only refer to diagram variables. Because of this restriction, in most cases if a hot condition is reached and evaluates to false, we know that it will never render true again. (Because of the partial order of events in an MSD, it could be that an assignment can be executed while the hot condition is enabled, and that this assignment changes the value of a diagram variable such that the condition expression can be evaluated to true. These cases are, however, not very common.)

Therefore, we can interpret it as a safety violation if a hot condition is enabled that evaluates to false. To encode this behavior, a pair of edges is created for each hot condition, similar to the encoding of cold conditions explained above. The update label of the violating edge additionally sets the variable `hotViolation` to true.

4.3.4 Forbidden messages

Forbidden messages are used to express that certain events must not occur while an MSD is active or that the active MSD shall be interrupted at the occurrence of a certain event (see Sect. 3.1.16). They must have the `Forbidden` stereotype applied and are allowed only at the end of an MSD. Furthermore, immediately prior to the first forbidden message in the MSD there must be a condition marking the end of the MSD. This condition must have the expression `false` and it must cover all lifelines of the MSD. The assumption `MSD Last-BreakBeforeEnterNext` is an example of an MSD that contains a cold forbidden message (see Fig. 4.2).

A hot forbidden message represents an event that must not occur while the MSD is active. Therefore, if the MSD is active and the forbidden event occurs, this is a safety violation. The only exception is that another, non-forbidden message representing the same event is contained in the MSD and is currently enabled. A cold forbidden message represents an event that if it occurs leads to a cold violation of the MSD. Again, if there is currently a regular message enabled which represents that event, this does not lead to a cold violation of the MSD. Note that the occurrence of an event that corresponds to a cold forbidden message in an MSD does not produce a safety violation if the active MSD is in a hot cut. In other words, the occurrence of an event that corresponds to a cold forbidden message always safely interrupts the active diagram.

In the MSD automaton template, forbidden messages are not represented by any edge. Instead they just affect the return expression of the local functions `eventInMSD(int ev)` and `hotEventEnabled()` of the MSD automaton template. These functions influence whether the edges for the cold violation or the hot (safety) violation are taken in the MSD automaton (see Fig. 4.4). These functions are extended for each cold and hot forbidden messages in the MSD as follows

First, for each hot or cold forbidden message, the return statement of the function `eventInMSD(int ev)` is extended to return true if the given parameter corresponds to the forbidden message. Second, the function `hotEventEnabled()`

is extended to always return false if the current event corresponds to a cold forbidden message and to always returns true if the current event corresponds to a hot forbidden message and the MSD is active, i.e., the lifeline variables are not in the initial configuration. If the current event neither corresponds to a hot or cold forbidden message, the function still returns whether the current cut is hot.

To realize this behavior of the function `hotEventEnabled()`, the return statement is constructed according to the following schema (see Listing 4.7). The return statement is a conjunction that as a first statement contains a disjunction. The first statement in the disjunction checks whether any hot event is enabled in the current cut of the active MSD. The disjunction furthermore consists of statements that check whether the current event corresponds to a hot forbidden message and if not all lifeline variables are at the initial configuration. The second statement of the conjunction checks whether the current event does not correspond to a cold forbidden message.

Listing 4.6: The schema for constructing the function `hotEventEnabled()`

```

/* returns whether any hot event is enabled in the
   current cut of the MSD */
bool hotEventEnabled(){
    return ((<a hot message is enabled in the current cut>
            or (<the current event corresponds to a hot
                forbidden message>
                and <the lifeline variables are not at the
                    initial configuration >))
            and <the current event does not correspond to a
                cold forbidden message>);
}

```

4.3.5 Assumption MSDs

The example MSD specification introduced above (see Fig. 4.2) contains an assumption MSD. Assumption MSDs have the stereotype `EnvironmentAssumption` applied, but otherwise the same constructs are used as in requirement MSDs.

The set of assumption MSDs describes the possible sequences of events that can take place in the environment. Arbitrary sequences of events may occur in the environment as long as, first, never a hot violation occurs in the assumption MSDs and, second, eventually there are no active (enabled executed) events in any of the active assumption MSDs. The system is only obliged to adhere to the requirements if no assumption MSD is violated.

The encoding of the assumption MSDs basically follows the same principle as the encoding of the requirement MSDs. There are just a few differences. First, all the edges in the assumption MSD automaton are uncontrollable. Second, edges which represent hidden events are synchronized over the binary channel `hiddenAssumptionMSDEvent` instead of the channel `hiddenEvent`. Third, if a safety violation occurs, this results in setting the variable `hotEnvViolation` to true instead of the variable `hotViolation`.

Figure 4.7 shows the automaton created for the MSD `LastBreakBeforeEnterNext`. The edges are uncontrollable and the edges that represent the hidden events (the hot condition and the end of the MSD) are synchronized via the channel `hiddenAssumptionMSDEvent`. The edge that represents the violation of the hot condition sets the variable `hotEnvViolation` to true. Aside the just highlighted exceptions, the encoding of the different parts of the assumption MSD follows the principles explained in Sect. 4.3.2. Again, not shown in this figure is that all edges that loop in the initial location set the variables `sysCutChanged` and `envCutChanged` to true, in order to express that an event produced by the system or environment automaton has changed the cut of at least one MSD in the specification (see Sect. 4.3.1).

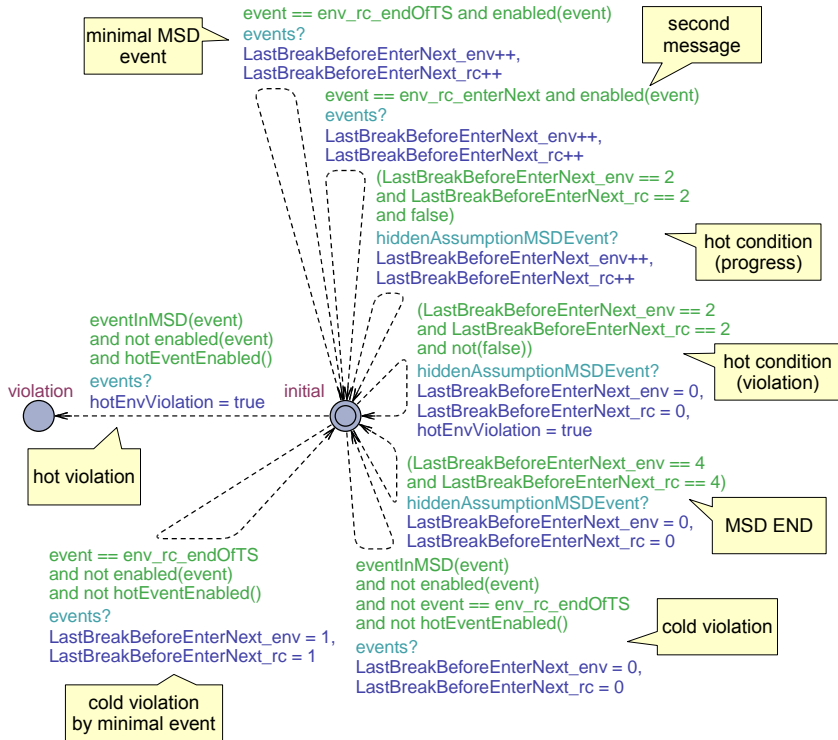


Figure 4.7: The TGA for the assumption MSD `LastBreakBeforeEnterNext`

Listing 4.7 shows the functions that are created locally for the MSD automaton template. The encoding of these functions follows the principles explained above (Sect. 4.3.2). Note that the return statement of the function `eventInMSD(int ev)` returns true if the given parameter value equals the constant representing the forbidden event `lastBreak`. Also the conjunction in the return statement of the function `hotEventEnabled()` contains the statement `event != env_rc_lastBreak`. Thus the function always returns false if the current event is `lastBreak`. This encoding ensures that the cold violation edge is taken if that event occurs (and is not currently enabled).

Listing 4.7: The functions declared in the automaton template for the MSD LastBreakBeforeEnterNext

```

/* returns whether the event ev is enabled in the
   current cut of the MSD */
bool enabled(int ev){
    return (ev == env_rc_endOfTS
            and LastBreakBeforeEnterNext_env == 0
            and LastBreakBeforeEnterNext_rc == 0
            or (ev == env_rc_enterNext
                and LastBreakBeforeEnterNext_rc == 1
                and LastBreakBeforeEnterNext_env == 1)
    );
}

/* returns whether any hot event is enabled in the
   current cut of the MSD */
bool hotEventEnabled(){
    return ((false
            or false)
            and event != env_rc_lastBreak);
}

/* returns whether the event ev is in the MSD */
bool eventInMSD(int ev){
    return (ev == env_rc_endOfTS
            or ev == env_rc_enterNext
            or ev == env_rc_lastBreak);
}

```

The encoding explained above produces a TGA network that represents the play-out behavior of the MSD specification. The difference to the original semantics of play-out is that here an iterative semantics is assumed (see Sect. 3.1.6) and that the concept of assumption MSDs is introduced.

4.4 The winning condition

In order for UPPAAL TIGA to calculate an admissible controller for the MSD specification, it remains to specify a winning condition that formulates when the system satisfies the MSD specification. In the following, a temporal property is explained by which UPPAAL TIGA checks the *consistent executability* (see Sect. 3.1.9) of the MSD specification under consideration of environment events. In the discussion (Sect. 4.7) it will be explained how to slightly modify the system and environment automata so that also the *consistency* of the specification can be shown by UPPAAL TIGA.

4.4.1 Checking consistent executability with Uppaal TIGA

As explained in Sect. 3.1.9, a specification is consistently executable if there exists a play-out algorithm (as the implementation of the system) which never produces any safety or liveness violations, i.e., never takes forbidden steps and

always eventually takes steps in order to progress beyond enabled hot messages and conditions. Additionally, the environment must be able to produce events infinitely often, or, in other words, the system must “listen” to environment events infinitely often and never take an infinite super-step. When extending the setting by environment assumptions in the form of assumption MSDs, the system is only required to fulfill the requirements if the environment assumptions are satisfied. The environment assumptions are satisfied if there are never any safety or liveness violations in the assumption MSDs. In contrast to the system, we do not require the environment to be inactive infinitely often. This is already implied by the play-out setting: the environment has to expect system events to occur after each environment event.

The winning condition

This leads to a winning condition for the system which intuitively requires that

- the system listens for environment events infinitely often
- and*
- all requirement MSDs are inactive infinitely often, i.e., infinitely often there are no active MSDs with enabled executed events
 - there is never a safety violation in any requirement MSD
- unless*
- not all assumption MSDs are inactive infinitely often or
 - there is a safety violation in any of the assumption MSDs

Note that we always require that the system eventually listen for environment events again. Otherwise, the system could easily force the environment to violate the assumptions by keeping the environment from progressing beyond enabled executed messages in active assumption MSDs.

In UPPAAL TIGA, this can be formalized as a CTL formula with respect to a TGA network of the above kind as

$$\begin{aligned}
 &AG(AF(\textit{systemProcess.systemInactive and} \\
 &\quad (\textit{not isActive(sys)} \\
 &\quad \textit{and not hotViolation} \quad (4.1) \\
 &\quad \textit{or hotEnvViolation} \\
 &\quad \textit{or isActive(env))))
 \end{aligned}$$

This formula states that the TGA network must be infinitely often in a state where the following conditions hold. First, the system automaton (its running instance is called *systemProcess*) is in the location *systemInactive* (here the system listens for environment events). Second, there are no events active in any requirement MSD. Third, no safety violation occurred in any requirement MSD. Alternatively, there occurred a safety violation in an assumption MSD or there is an active event in an assumption MSD. The latter two conditions mean that the system automatically “wins” if the environment does not avoid a safety violation and not all assumption MSDs are inactive infinitely often.

4.4.2 An alternative winning condition

In a previous publication of the synthesis approach [Gre10], a simpler winning condition was proposed, which simply states that safety violations must never occur. Larsen et al. propose a similar condition [LLNP10]. Under certain circumstances such a property is sufficient and UPPAAL TIGA can find an admissible controller for the specification more efficiently. Therefore, in cases where the runtime or the space required by the synthesis is critical, this winning condition may be used instead of the one presented above (see the different benchmarks documented in Appendix C).

When considering environment assumptions, a winning condition can be specified that intuitively states that it must not be the case that a state is reached where the following conditions hold

- a safety violation occurred in a requirement MSDs
- no safety violation occurred in any assumption MSD
- the environment is inactive, i.e., it has managed to progress beyond all active events and it has processed all hidden events

The latter condition is important to keep the environment from provoking a safety violation in a requirement MSD when this inevitably leads to a safety violation of an assumption MSD. For example, the environment may violate the requirements by producing an event which leads an assumption MSD in a cut where a hot `false` condition is enabled. The above property states that the environment can only win if it processes this enabled hidden event. In this case, however, this sets the `hotEnvViolation` variable to true and the environment loses. (Remember that `hotEnvViolation` is never reset to true again.)

This winning condition can be specified in CTL as

$$\begin{aligned}
 &AG(\text{not } ((\text{systemProcess.systemActive} \\
 &\text{and environmentProcess.environmentInitial}) \\
 &\text{and hotViolation} \tag{4.2} \\
 &\text{and not hotEnvViolation} \\
 &\text{and not isActive(env)}))
 \end{aligned}$$

This formula states that it must never be the case that the system and environment automaton are in the locations `systemActive` resp. `environmentInitial`, that a safety violation occurred in a requirement MSD, that no safety violation has occurred in any assumption MSD, and that all the assumption MSDs are inactive.

The above property, in the following called the *AG property*, is weaker than the previous property in (4.1), which is called *AGAF property* in the following. By the *AG* property, the system may also win by being active forever from some point on. Therefore, this property may only be used if it can be guaranteed that the specification is free of potential loops in which the system can always keep at least one MSD active. In larger specifications, such loops may be difficult to detect. Therefore, to ensure that a controller synthesized via the *AG* property

is an admissible implementation of the specification, a mechanism for automatically checking the absence of such loops in the controller would need to be developed in the future.

4.5 Mapping timed MSD specifications

This section explains the mapping from a timed MSD specification to a network of TGA in UPPAAL TIGA. In a timed setting, some details change in the environment and the system automaton templates that are created for a specification. These changes are explained in Sect. 4.5.1. The principle for encoding requirement or assumption MSDs as presented above (Sect. 4.3.2 and 4.3.5) does not change in a timed setting. The mapping is only extended by the encoding of clock resets and time conditions. These extensions are explained in Sect. 4.5.2. The winning condition presented above is extended slightly in a timed setting. This extension is explained in Sect. 4.5.3.

The encoding principles are explained by the help of MSDs from the example specification shown in Fig. 4.8. This specification captures the combined requirements and assumptions from the use case *Drive onto track section* and *Drive onto merging switch*, based on the use case descriptions shown in Fig. 2.3 and Fig. 2.4. The requirements and assumptions from the MSD specification in Fig. 4.2 recur in this specification with slight modifications. First, the MSDs *RequestEnterAtEndOfTrackSection* and *ReplyBeforeLastSafeBreak* of Fig. 4.2 are merged into the extended MSD *RequestEnterAtEndOfTrackSection* that is shown here. Second, the assumption MSD *LastBreakBeforeEnterNext*, which was formulated as an anti-scenario in Fig. 4.2, is replaced by the MSD *PassingTrackSectionPointsInSequence*. Additionally, the MSD specification contains the MSD *ReplyOfSwitchControlNotTooEarly*, which was already explained in Sect. 3.1.14 (see Fig. 3.8). Also this specification contains the assumption MSDs *LastBreakDelay* and *EndOfTSToEnterNextDelay*. The MSD *LastBreakDelay* expresses that the `lastBreak` event does not occur less than five seconds after the event `endOfTS`. The MSD *EndOfTSToEnterNextDelay* says that the event `enterNext` occurs within eight to twelve seconds after the event `endOfTS`.

4.5.1 The environment and system automata for timed specifications

Remember that for the play-out of LSC/MSD specifications with an untimed setting, we assume that the system actions take no time and that the system is always fast enough to perform any finite number of steps before the occurrence of the next environment event (synchrony hypothesis) [HM03]. In a timed setting, we stick with that assumption. Time can only pass between events, and environment events occur only when the system is waiting.

In the timed play-out described by Harel and Marelly [HM02b], the play-out always immediately executes active events. The play-out only delays steps if it is forced to wait before minimal delays, i.e., hot time conditions that specify a lower time bound.

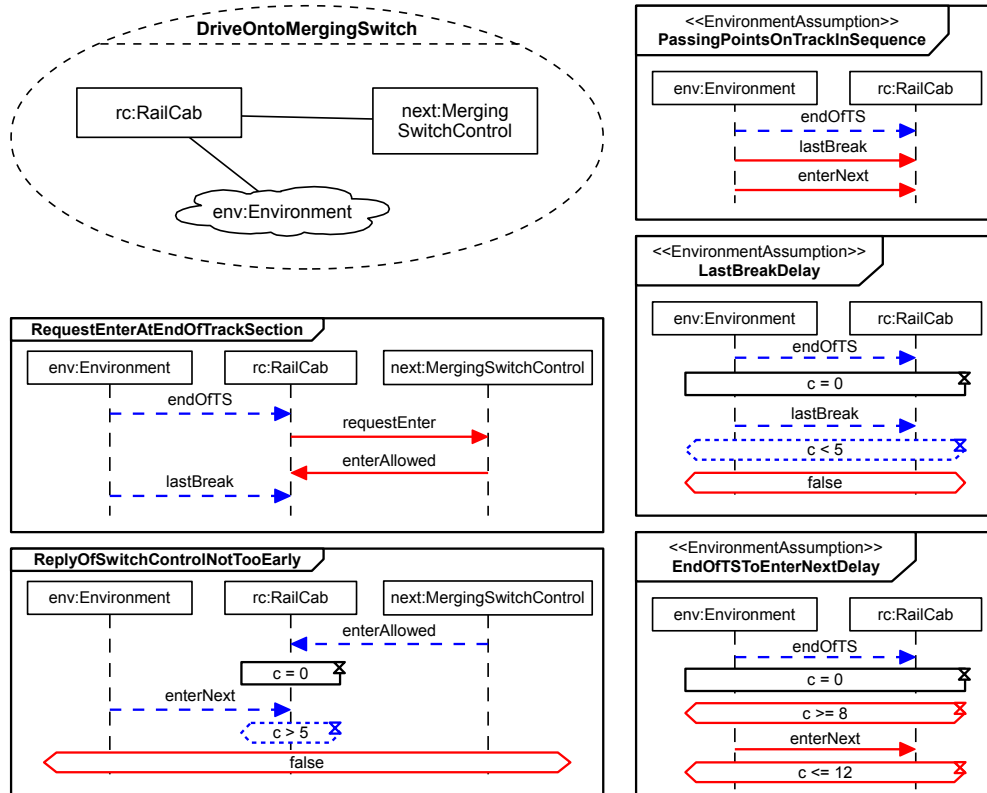


Figure 4.8: The specification of the use case Drive onto merging switch as an example to illustrate the MSD-to-TGA mapping for timed specifications

However, in the example use case Drive onto merging switch shown above, the system can only adhere to the specification if the track section control delays its reply `enterAllowed` (see the explanations for the MSD `ReplyOfSwitchControlNotTooEarly` in Sect. 3.1.14). In the synthesis, we therefore want to allow the system to delay any system step at any time, not only if minimal delays force the system to delay the next step. To encode this in the TGA system, there are a number of modifications to the environment and system automata in a timed setting compared to the previously described environment and system automata for the untimed setting.

This extra degree of freedom comes with a cost. Every time that the system waits, environment events may occur, which greatly increases the state space of the synthesis problem. For many examples, there exists a winning strategy where the system does not have to delay active events. To find these strategies, it is sufficient to only consider the restricted behavior, where the system may only wait when minimal delays force it to wait [HM02b]. This can greatly reduce the complexity of the synthesis problem. How to encode this restriction for specifications where the restriction is reasonable will be discussed in Sect. 4.7.1.

Let us look at environment and system automata for the timed setting in Fig. 4.9. These automata are very similar to those for untimed MSD specifications (see Fig. 4.3). Both automata can produce environment resp. system

events using the same two-step principle of first assigning a value to the variable `event` and then emitting over the broadcast channel events. Also, both automata have to process the enabled hidden events before simultaneously returning to the locations `environmentInitial` resp. `systemActive`.

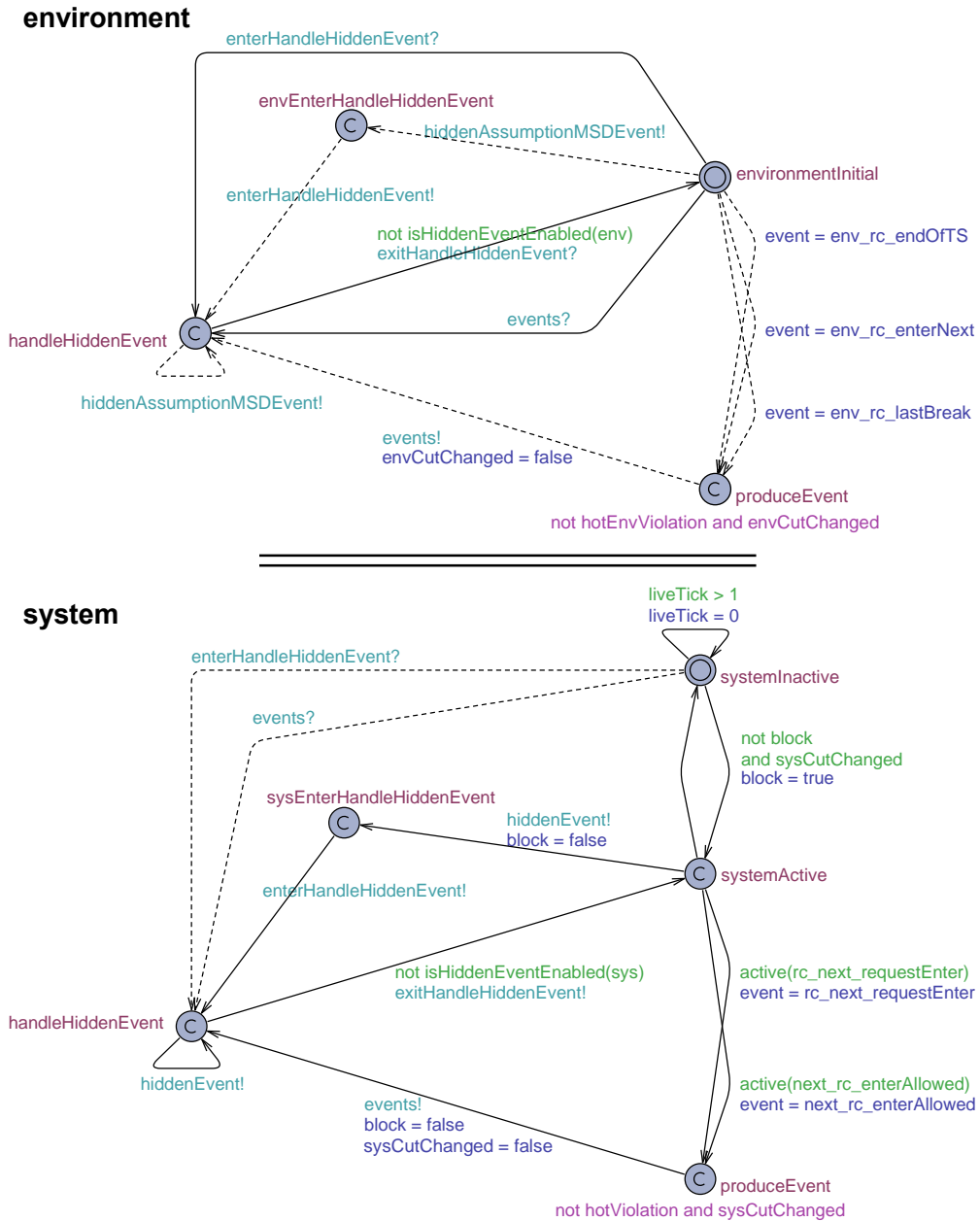


Figure 4.9: The environment and system automata for a timed example MSD specification

The extensions discussed in the following are introduced because the system may choose to wait or it may be forced to wait by hot minimal delays. When

waiting, the system must be able to “return to action” without a triggering environment event, i.e., the system must be able to send a message after a certain time has passed or it must be able to trigger the progress beyond a hot minimal delay that has rendered true after some time has passed.

Processing minimal delays

One change in these automata, compared to those for the untimed setting, is that, when the automata are in the locations `environmentInitial` resp. `systemActive`, the environment or system can process hidden events by taking an edge to the location `envEnterHandleHiddenEvent` resp. `sysEnterHandleHiddenEvent`. This edge emits over the binary channel `hiddenAssumptionMSDEvent` resp. `hiddenEvent`. This extension is necessary because in timed MSD specifications, the MSDs and assumption MSDs may contain minimal delays. In the above example specification, there is for example the hot time condition $c \geq 8$ in the assumption MSD `EndOfTSToEnterNextDelay` shown in Fig. 4.8. Such minimal delays require that the cut does not progress in the active MSD before the time condition evaluates to true. Therefore, the environment or system must wait until the condition becomes true before progressing the cut.

According to the original semantics of LSC, the cut must immediately progress beyond a minimal delay if it is enabled and evaluates to true [HM02b, HM03]. This is the same behavior as required for all enabled hidden events. However, in this encoding, the system resp. environment is not forced to progress enabled minimal delays that evaluate to true immediately. Minimal delays are therefore an exception to the general rule that enabled hidden events must be processed immediately. This exception is made because it is not possible in the timed automata of UPPAAL to force a transition immediately when a clock variable is greater a given value. For example, imagine that the condition of the minimal delay in the MSD `EndOfTSToEnterNextDelay` was $c > 8$. Then it would not be possible to say “wait until c is greater than 8, but also do not wait any longer”.

When either the system or environment takes the edge to the location `envEnterHandleHiddenEvent` resp. `sysEnterHandleHiddenEvent`, they must leave this location immediately to the location `handleHiddenEvent`. The reason for this is that, after processing a minimal delay, other hidden events may be enabled that need to be processed immediately. The edge from `envEnterHandleHiddenEvent` resp. `sysEnterHandleHiddenEvent` to `handleHiddenEvent` also emits over the binary channel `enterHandleHiddenEvent`. The result is that always both automata simultaneously end up in the location `handleHiddenEvent` from where they eventually simultaneously progress to the locations `environmentInitial` and `systemActive`.

Priority to system events

As stated above, we assume that time may only pass between events. Therefore the system can take any finite number of steps before the next environment

event occurs, provided that it does not choose to wait or is forced to wait (by a minimal delay). This semantics is encoded as follows.

Just like in the system automaton for the untimed setting, the location `systemActive` is committed. Since enabled edges leaving a committed location must be taken before any other edge is taken, the system now has the priority in deciding to produce one of the currently enabled events by taking an edge to the location `produceEvent`. Alternatively, it can also choose to wait and listen for another environment event to occur by taking an edge to the location `systemInactive`. If no environment event occurs and the system decides to take the next step after some time has passed, it can switch to the location `systemActive` again and produce events from there. If the system chooses to wait and an environment event occurs, the system automaton progresses to the location `handleHiddenEvent` and eventually ends up in the location `systemActive`. It then has the chance to produce system events before the next environment event occurs.

One effect that must be avoided in the system automaton is that the system loops between the locations `systemActive` and `systemInactive` infinitely often. If that was possible, the system could take infinitely many transitions in zero time while the environment must wait to produce events in order not to violate any assumption. This way, the system could in some cases be able to satisfy the winning condition, because no time will pass anymore and the environment will never be able to produce another event.

To avoid this behavior, the Boolean variable `block` is introduced. This variable is set to true if the system decides to move from the location `systemInactive` back to the location `systemActive`. If it decides to immediately return to `systemInactive`, it will not be able to take the edge to `systemActive` another time. This is ensured by the guard `not block`. The variable `block` is only set to false again when the system produces an event or processes a hidden event.

4.5.2 Encoding clock resets and time conditions

As mentioned above, when mapping timed MSDs to the corresponding TGA, the principles for encoding messages, assignments, and conditions as explained for untimed MSD specifications (see Sect. 4.3.2) remain unchanged. The mapping must only be extended by the encoding of clock resets and time conditions.

The encoding of clock resets and time conditions is explained in the following by the help of the example MSD `ReplyOfSwitchControlNotTooEarly` (see Fig. 4.8). The MSD automaton corresponding to this MSD is shown in Fig. 4.10. The contributions to the global declarations for the MSD are shown in Listing 4.8 and the declarations created locally for the MSD in the MSD automaton template are shown in Listing 4.9. Again not shown is that all edges looping in the initial location set the variables `envCutChanged` and `sysCutChanged` to true.

Clock resets

Let us have a look at the MSD automaton in Fig. 4.10. The encoding of the minimal message event and the second message event follow the same principle

as explained previously in Sect. 4.3. Also the edges for a hot violation, the two cases for the cold violation and the edge representing the termination of the MSD are created as explained above. The clock reset is encoded like a normal assignment except that the assigned variable is a clock variable, see Listing 4.9.

Cold time conditions

The example furthermore shows the encoding of a cold time condition. As explained in Sect. 3.1.14, when the cut reaches a cold time condition in an MSD, it is immediately evaluated. When it evaluates to true, the cut progresses beyond the condition, when the cut evaluates to false, a cold violation occurs. Thus, a cold time condition is encoded with the same progressing/violating edge pair as an untimed cold condition. The progressing edge is enabled when the lifeline variables of the lifelines covered by the time condition are at the position immediately before the time condition and when the time constraint evaluates to true. Then, the cut can progress. The violating edge is enabled when the lifeline variables are at the position immediately before the time condition, but the time constraint evaluates to false. Then, the cut is reset by resetting all lifeline variables to zero, which represents the effect of a cold violation.

For each cold time condition, also the implementation of the function `<MSD-name>_hiddenEventEnabled()` is extended such that it returns true when the lifeline variables of the lifelines covered by the time condition are at the position immediately prior to the time condition (see Listing 4.8).

Note that in UPPAAL, one cannot simply form negations of time expressions (as it is done in the case of untimed cold conditions). Therefore, to negate the time constraint, the mapping must invert the compare operator, e.g. instead of writing `not (c > 5)`, the expression must be rewritten to `c <= 5`. Furthermore, expressions that test for the equality of a clock variable, like `c == 5` are prohibited in cold conditions because expressions like `not (c == 5)` or `c ≠ 5` are not allowed as guard expressions in UPPAAL. This approach requires that the expressions in cold time constraints are given in the form $c_1 \bowtie \langle \text{expr} \rangle$ and $c_1 - c_2 \bowtie \langle \text{expr} \rangle$ where c_1 and c_2 are clock variables, \bowtie is an operator $\{<, \leq, \geq, >\}$, and $\langle \text{expr} \rangle$ is an expression evaluating to an integer value. Furthermore, the only variables allowed in $\langle \text{expr} \rangle$ are MSD variables defined by assignments prior to the condition.

The global declarations shown in Listing 4.8 are created based on the encoding principles explained in Sect. 4.3, and are included here merely to give a complete picture of the encoding of the MSD `ReplyOfSwitchControlNotTooEarly`. The listing first shows the lifeline variables that are created for the lifelines in the MSD. Second, the function `ReplyOfSwitchControlNotTooEarly_active(int ev)` is shown. It only returns false, because this MSD does not contain any executed messages. Third, the function `ReplyOfSwitchControlNotTooEarly_isHiddenEventEnabled()` is shown. It contains an expression that returns true if the cut has reached the end of the MSD (the lifeline variables are in the configuration (3,5,2)), if the cut is in front of the hot false condition (the lifeline variables are in the configuration (2,4,2)), if the cut is in front of the cold condition (the lifeline variable for the lifeline `rc` equals 3), or if the cut is in front

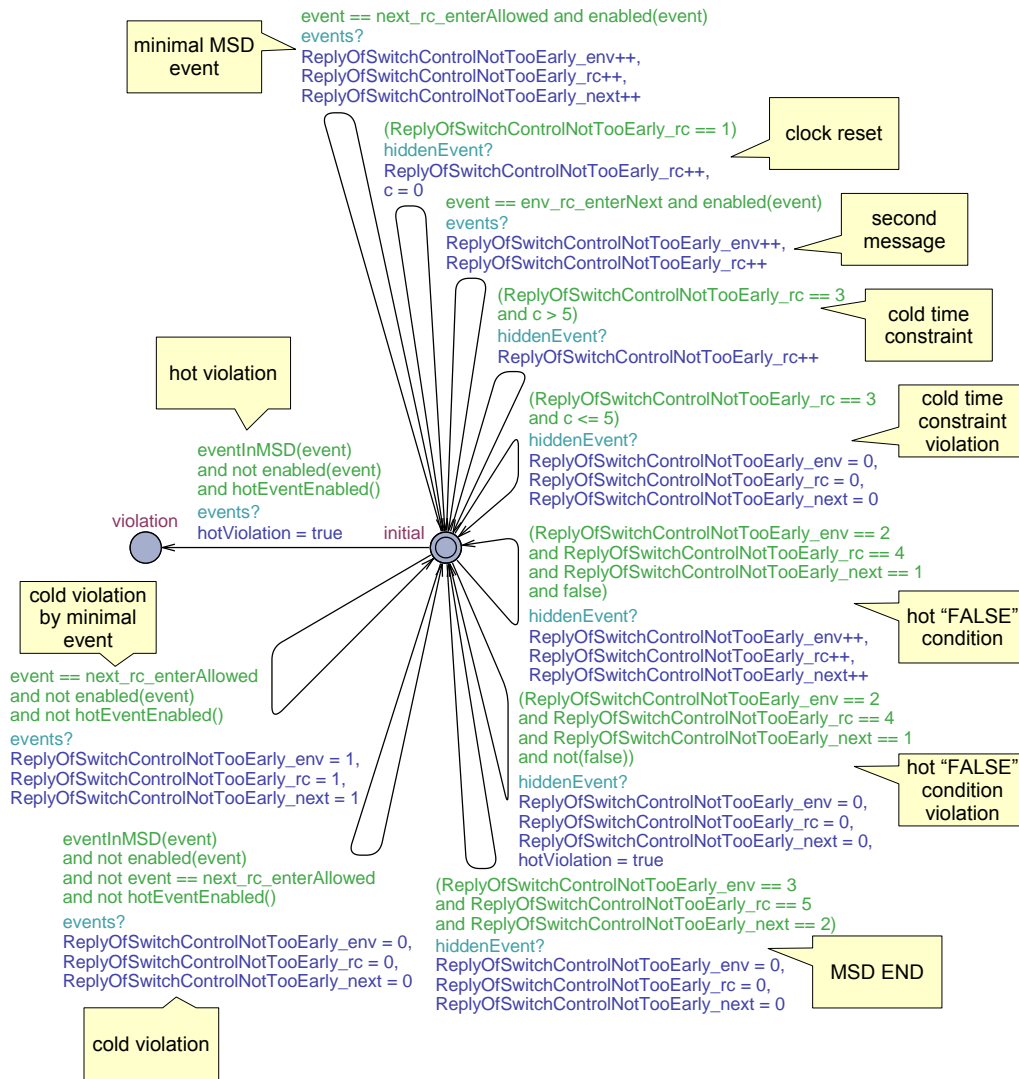


Figure 4.10: The automaton template for the MSD ReplyOfSwitchControlNotTooEarly

of the assignment (the lifeline variable for the lifeline rc equals 1). Furthermore, the listing shows how the functions `isHiddenEventEnabled(int player)`, `active(int ev)`, and `isActive(int player)` are extended by referring to the above functions.

Listing 4.8: Global declarations for the MSD `ReplyOfSwitchControlNotTooEarly`

```

...

int ReplyOfSwitchControlNotTooEarly_env = 0;
int ReplyOfSwitchControlNotTooEarly_rc = 0;
int ReplyOfSwitchControlNotTooEarly_next = 0;

...

/* return whether the event ev is active in the
   current cut of MSD ReplyOfSwitchControlNotTooEarly */
bool ReplyOfSwitchControlNotTooEarly_active(int ev){
    return false; // The MSD contains no executed events...
}

...

/* return whether any hidden event is enabled in the
   current cut of MSD ReplyOfSwitchControlNotTooEarly */
bool ReplyOfSwitchControlNotTooEarly_isHiddenEventEnabled(){
    return ((ReplyOfSwitchControlNotTooEarly_env == 3
              and ReplyOfSwitchControlNotTooEarly_rc == 5
              and ReplyOfSwitchControlNotTooEarly_next == 2)
            or ReplyOfSwitchControlNotTooEarly_env == 2
              and ReplyOfSwitchControlNotTooEarly_rc == 4
              and ReplyOfSwitchControlNotTooEarly_next == 1
            or ReplyOfSwitchControlNotTooEarly_rc == 1
            or ReplyOfSwitchControlNotTooEarly_rc == 3);
}

...

/* returns whether there is any hidden event enabled in a
   requirement MSD (player = sys) or assumption MSD
   (player = env) */
bool isHiddenEventEnabled(int player){
    return ( ...
            or (ReplyOfSwitchControlNotTooEarly_isHiddenEventEnabled()
                and player == sys)
            or ... );
}

/* returns whether there is any executed message enabled in
   any of the MSDs that represents the given diagram event */
bool active(int ev){
    return ( ...
            or ReplyOfSwitchControlNotTooEarly_active(ev)
            or ... );
}

/* returns whether there is any executed messages or any

```

```

    hidden event is enabled in a requirement MSD (player = sys)
    or assumption MSD (player = env) */
    bool isActive(int player){
    return (isHiddenEventEnabled(player)
           or ...
           or (
             RequestEnterAtEndOfTrackSection_active(next_rc_enterAllowed)
             and player == sys));
    }

```

Listing 4.9 shows the implementations of the functions declared locally for the MSD template automaton created for the MSD `ReplyOfSwitchControlNotTooEarly`. Also the declaration of the clock variable `c` is shown. The function `enabled(int ev)` returns true if the lifeline variables are in a configuration where the message `enterNext` is enabled and when called with the respective value for the `ev` parameter. It also returns true if the lifeline variables are in the initial configuration, where the first message `enterAllowed` is enabled, and when called with the respective value for the `ev` parameter. The function `hotEventEnabled()` always returns false because there is no hot message nor hot minimal delay in the MSD. The function `eventInMSD(int ev)` returns true if it is called with a value for the parameter that represents a message that appears in the MSD, which is `enterAllowed` and `enterNext` in this case.

Listing 4.9: Local declarations of the MSD automaton template for the MSD `ReplyOfSwitchControlNotTooEarly`

```

clock c;

/* returns whether the event ev is enabled in the
   current cut of the MSD */
bool enabled(int ev){
return (
    (ev == env_rc_enterNext and
     ReplyOfSwitchControlNotTooEarly_rc == 2 and
     ReplyOfSwitchControlNotTooEarly_env == 1)
    or (ev == next_rc_enterAllowed and
        ReplyOfSwitchControlNotTooEarly_rc == 0 and
        ReplyOfSwitchControlNotTooEarly_next == 0 and
        ReplyOfSwitchControlNotTooEarly_env == 0
    );
}

/* returns whether any hot event is enabled in the
   current cut of the MSD */
bool hotEventEnabled(){
return false; // no hot msg or hot min delay in this MSD...
}

/* returns whether the event ev is in the MSD */
bool eventInMSD(int ev){
return (ev == env_rc_enterNext
       or ev == next_rc_enterAllowed
    );
}

```

Hot time conditions

Hot time conditions can specify minimal and maximal delays in the MSDs. Minimal delays specify lower time bounds of the form $c \geq \langle \text{expr} \rangle$ or $c > \langle \text{expr} \rangle$ and maximal delays specify upper time bounds of the form $c < \langle \text{expr} \rangle$ or $c \leq \langle \text{expr} \rangle$. Again $\langle \text{expr} \rangle$ must be an UPPAAL-compliant expression evaluating to an integer value. Thus far, the synthesis supports only expressions formed from integer constants and integer variables that were defined by assignments preceding the condition.

Due to this restriction, if a *maximal delay* is enabled, but its expression evaluates to false, it is possible to immediately determine that a hot violation has occurred, because it will never be able to become true if more time elapses. Therefore, hot maximal delays are encoded similarly to untimed hot conditions by a pair of a progressing and a violating edge, where the violating edge sets the variable `hotViolation` or `hotEnvViolation` (in the case of the assumption MSD) to true. The return statement of the global function `<MSD-name>_hiddenEventEnabled()` is extended such that it returns true when the cut is prior to the maximal delay.

Minimal delays are conditions where the expression will always render true after some time. Therefore, each hot minimal delay is encoded by only one edge in the MSD automaton that can progress the cut when the condition expression evaluates to true. For minimal delays, the implementation of the function `<MSD-name>_hiddenEventEnabled()` must however not evaluate to true when the cut is immediately prior to the hot minimal delay. This would force the system or environment to immediately progress beyond the condition although it may not yet be possible. It would be desirable to also include the delay expression in the implementation of the `<MSD-name>_hiddenEventEnabled()` function. This way, we could force the system and environment automata to progress beyond enabled minimal delays as soon as they evaluate to true. That, however, is not possible since UPPAAL and UPPAAL TIGA do not allow clock variables to be used inside function bodies.

Therefore the implementation of the function `<MSD-name>_hiddenEventEnabled()` is not extended to return true when the cut is prior to a minimal delay. This means that, as soon as the delay expression evaluates to true, the system or environment can choose to progress the cut beyond the hot minimal delay at any time they please. See above (Fig. 4.9) how the environment and system automata that are created for the timed MSD specification have edges leaving the locations `environmentInitial` and `systemActive` to do so. As already discussed above (see p. 85), it is not possible in UPPAAL to encode a behavior that forces the environment or system to progress minimal delays immediately when they evaluate to true. If the expression is for example $c > 8$, it would not be possible to say “wait until c is greater than 8, but also do not wait any longer”.

If a minimal delay is enabled, i.e., the cut resides prior to the minimal delay on all lifelines that the condition covers, the cut is hot. To encode this, the body of the function `hotEventEnabled()` for the respective MSD automaton template

is extended for each minimal delay such that it returns true when the cut is immediately prior to the minimal delay.

The environment or system must eventually progress a hot cut, otherwise this is a liveness violation of the respective MSD. In order to express this, first, an additional function is created in the TGA model for timed MSD specifications that returns true if there is a minimal delay enabled in any assumption or environment MSD. Second, the winning condition is extended to express that the environment or system will not be able to win if there is any minimal delay enabled in any assumption or requirement MSD, respectively. (This will be explained shortly in Sect. 4.5.3.)

The additional global function that encodes whether a minimal delay is enabled in an assumption or requirement MSD is called `isMinimalDelayEnabled(int player)`. Called with the parameter `env`, this function returns whether there is a minimal delay enabled in an assumption MSD. Called with the parameter `sys`, this function returns whether there is a minimal delay enabled in a requirement MSD.

In the example specification shown in Fig. 4.8, the assumption MSD `EnterNextAfterEightToTwelveSeconds` contains a minimal and a maximal delay. Figure 4.11 shows the automaton created for that MSD. Listing 4.10 shows the extensions to the global declarations and globally declared functions that are made for this MSD. Listing 4.11 shows the local declarations in the MSD automaton template. The MSD template automaton and the listings are briefly described in the following. The extension to the winning condition in the timed setting is explained in Sect. 4.5.3.

The MSD template automaton for the MSD `EnterNextAfterEightToTwelveSeconds` as shown in Fig. 4.11 shows the encoding for different MSD constructs that were already explained before: the edges created for the minimal event (first message), the second message, the termination of the MSD, the cold violation, the cold violation by the minimal event, and the hot violation. The figure shows the one edge that is created for the minimal delay. The edge is enabled if the cut is prior to the condition on all lifelines that it covers, and the condition expression `c >= 8` must evaluate to true. The maximal delay is encoded by two edges that are enabled if the cut is immediately in front of the condition on all lifelines that it covers and if the condition expression `c <= 12` evaluates to true or false, respectively. One edge represents that the cut progresses upon a successful evaluation of the condition. The other represents a hot violation that consequently will occur if the expression is not true.

The extensions to the global declarations that are made for the MSD `EnterNextAfterEightToTwelveSeconds` are shown below in Listing 4.10. It first shows the two lifeline variables that are created for the two lifelines in the MSD. Below it shows the Boolean function `EnterNextAfterEightToTwelveSeconds_active(int ev)` that returns true if the parameter `ev` represents the message `enterNext` and the message is currently enabled in the MSD. The function `EnterNextAfterEightToTwelveSeconds_isHiddenEventEnabled()` encodes whether there is currently a hidden event enabled. This is the case if the cut is either at the end of the MSD, in front of the clock reset, or in front of the maximal delay. The minimal delay is not considered in this function. Whether a minimal delay is enabled

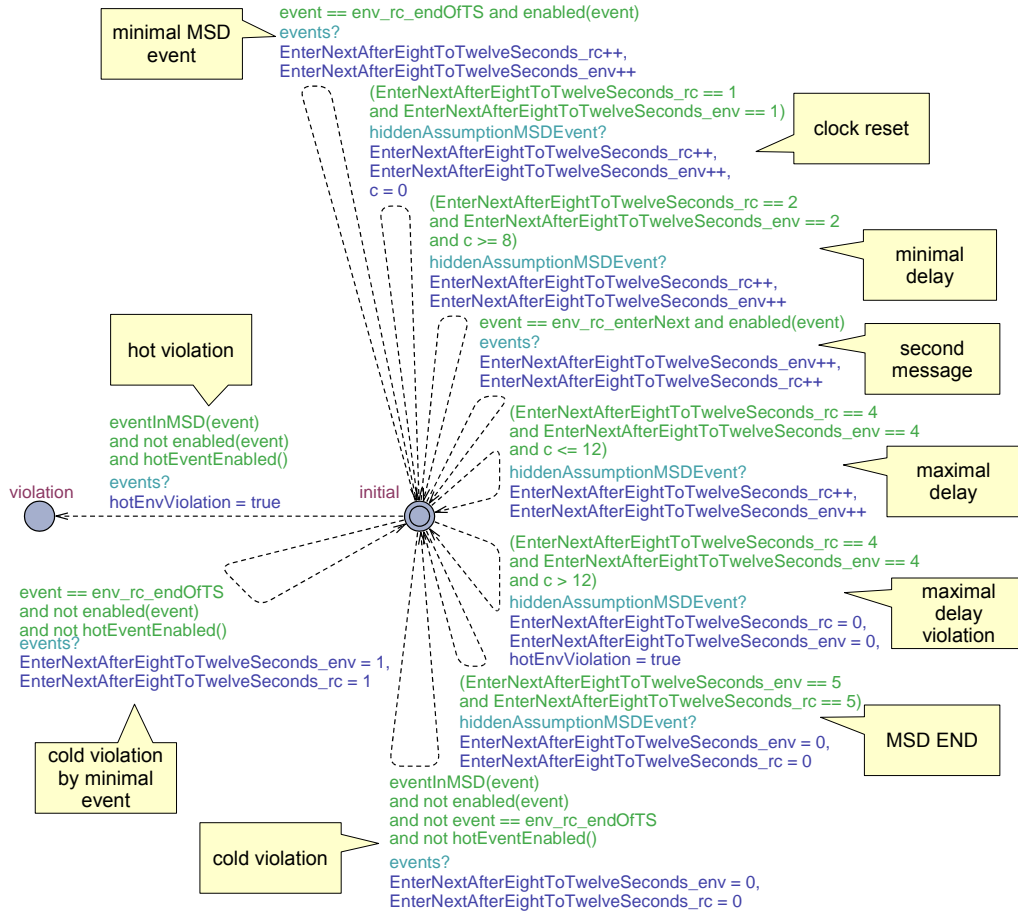


Figure 4.11: The automaton template for the assumption MSD `EnterNextAfterEightToTwelveSeconds`

in this MSD or in any other MSD is rather encoded in the function `isMinimalDelayEnabled(int player)`. Depending on the parameter, which can be either `env` or `sys`, the function returns whether there is any minimal delay enabled in any requirement or assumption MSD, respectively. In the use case specification `Drive onto merging switch` there is only one MSD with one minimal delay. Thus, the function returns true if the cut is prior to that minimal delay and, since the MSD is an assumption MSD, if it is called with the parameter `env`.

The Boolean function `isHiddenEventEnabled(int player)` is extended to return true if it is called with the parameter `env` and a hidden event is enabled in the assumption MSD `EnterNextAfterEightToTwelveSeconds`. Whether this is the case is answered by a call to the function `EnterNextAfterEightToTwelveSeconds_isHiddenEventEnabled()` (explained above) in the disjunction that makes up the function's return statement. The Boolean function `active(int ev)` is extended to return true if it is called with a parameter that represents a message event appearing in the MSD `EnterNextAfterEightToTwelveSeconds` that is currently active. This is done by including a call to the function `EnterNextAfterEight-`

ToTwelveSeconds_active(int ev) in the disjunction that makes up the function's return statement. Last, the Boolean function isActive(int player) is extended to return true if it is called with the parameter env and there is an active event in MSD EnterNextAfterEightToTwelveSeconds. For this purpose, a call to the above-mentioned function EnterNextAfterEightToTwelveSeconds_active(int ev) is included in the disjunction that makes up the function's return statement for each executed event that appears in the MSD EnterNextAfterEightToTwelveSeconds (which is only the message enterNext).

Listing 4.10: Global declarations for the MSD EnterNextAfterEightToTwelveSeconds

```

...

// the lifeline variables
int EnterNextAfterEightToTwelveSeconds_env = 0;
int EnterNextAfterEightToTwelveSeconds_rc = 0;

...

/* return whether the event ev is active in the
   current cut of MSD EnterNextAfterEightToTwelveSeconds */
bool EnterNextAfterEightToTwelveSeconds_active(int ev){
    return (false
            or (ev == env_rc_enterNext
                and EnterNextAfterEightToTwelveSeconds_env == 3
                and EnterNextAfterEightToTwelveSeconds_rc == 3));
}

...

/* return whether any hidden event (except minimal delays)
   is enabled in the current cut of MSD
   EnterNextAfterEightToTwelveSeconds */
bool EnterNextAfterEightToTwelveSeconds_isHiddenEventEnabled(){
    return ((EnterNextAfterEightToTwelveSeconds_env == 5
            and EnterNextAfterEightToTwelveSeconds_rc == 5)
            or EnterNextAfterEightToTwelveSeconds_rc == 1
            and EnterNextAfterEightToTwelveSeconds_env == 1
            or EnterNextAfterEightToTwelveSeconds_rc == 4
            and EnterNextAfterEightToTwelveSeconds_env == 4);
}

...

/* return whether there are any minimal delay enabled in the
   current cut of MSD EnterNextAfterEightToTwelveSeconds */
bool isMinimalDelayEnabled(int player){
    return (false
            or (player == env
                and EnterNextAfterEightToTwelveSeconds_rc == 2
                and EnterNextAfterEightToTwelveSeconds_env == 2));
}

...

```

```

/* returns whether there is any hidden event enabled in a
   requirement MSD (player = sys) or assumption MSD
   (player = env) */
bool isHiddenEventEnabled(int player){
    return ( ...
        or (
            EnterNextAfterEightToTwelveSeconds_isHiddenEventEnabled()
            and player == env)
        or ...);
}

/* returns whether there is any executed message enabled in
   any of the MSDs that represents the given diagram event */
bool active(int ev){
    return (...
        or EnterNextAfterEightToTwelveSeconds_active(ev));
}

/* returns whether there is any executed messages or any
   hidden event is enabled in a requirement MSD (player = sys)
   or assumption MSD (player = env) */
bool isActive(int player){
    return (...
        or
        (EnterNextAfterEightToTwelveSeconds_active(env_rc_enterNext)
        and player == env));
}

```

The declarations created locally for the automaton template of the MSD `EnterNextAfterEightToTwelveSeconds` are shown in Listing 4.11 below.

The Boolean function `enabled(int ev)` is created to return true if it is called with a parameter that represents an event that is represented by a message in the MSD that is currently enabled. In the initial cut (where both lifeline variables are zero), the first message `endOfTS` is enabled. The message `endOfTS` is enabled if both lifeline variables equal three. The Boolean function `hotEventEnabled()` is created to return true if the current cut is hot. This is the case where the message `enterNext` is enabled, but also if the minimal delay is enabled. Last, the function `eventInMSD(int ev)` is created to return true if a certain message event occurs in the MSD. This is the case for the message events `endOfTS` and `enterNext`. Thus the function returns true if called with a parameter which represents one of these events.

Listing 4.11: Local declarations of the MSD automaton template for the MSD `EnterNextAfterEightToTwelveSeconds`

```

clock c;

/* returns whether the event ev is enabled in the
   current cut of the MSD */
bool enabled(int ev){
    return (ev == env_rc_endOfTS
        and EnterNextAfterEightToTwelveSeconds_rc == 0
        and EnterNextAfterEightToTwelveSeconds_env == 0
        or (ev == env_rc_enterNext

```

```

        and EnterNextAfterEightToTwelveSeconds_rc == 3
        and EnterNextAfterEightToTwelveSeconds_env == 3));
    }

    /* returns whether any hot event is enabled in the
       current cut of the MSD */
    bool hotEventEnabled(){
        return (( false
            or (EnterNextAfterEightToTwelveSeconds_env == 3
                and EnterNextAfterEightToTwelveSeconds_rc == 3)
            or EnterNextAfterEightToTwelveSeconds_rc == 2
                and EnterNextAfterEightToTwelveSeconds_env == 2));
    }

    /* returns whether the event ev is in the MSD */
    bool eventInMSD(int ev){
        return (ev == env_rc_endOfTS
            or ev == env_rc_enterNext);
    }
}

```

4.5.3 Extensions to the winning condition for timed specifications

In a timed setting, the winning conditions presented in Sect. 4.4 have to be changed slightly in order to express that the system or the environment cannot win as long as there is a minimal delay enabled in a requirement or assumption MSD, respectively. For this purpose the winning conditions are extended with calls to the function `isMinimalDelayEnabled(int player)` as shown below.

The AGAF winning condition (4.1) is extended as follows.

$$\begin{aligned}
 &AG(AF(systemProcess.systemInactive \textit{and} \\
 &\quad (not \textit{isActive}(sys) \\
 &\quad \textit{and not hotViolation} \\
 &\quad \textit{and not isMinimalDelayEnabled}(sys) \qquad (4.3) \\
 &\quad \textit{or hotEnvViolation} \\
 &\quad \textit{or isActive}(env) \\
 &\quad \textit{or isMinimalDelayEnabled}(env)))
 \end{aligned}$$

It now states that the system wins if it is infinitely often in a state where the system automaton is in the location `systemInactive`, there is no active event in any requirement MSD, no hot violation has occurred in any requirement MSD, and there is no minimal delay enabled in any requirement MSD, *or* a hot violation has occurred in an assumption MSD, an environment event is active in an assumption MSD, or a minimal delay is enabled in an assumption MSD.

The AG winning condition (4.2) is extended as follows.

$$\begin{aligned}
 &AG(\text{not } ((\text{systemProcess.systemActive} \\
 &\text{and environmentProcess.environmentInitial}) \\
 &\quad \text{and hotViolation} \\
 &\quad \text{and not hotEnvViolation} \\
 &\quad \text{and not isActive(env)} \\
 &\quad \text{and not isMinimalDelayEnabled(env))))
 \end{aligned} \tag{4.4}$$

It now states that the system wins if it can always avoid a situation where the system and environment automaton are in the locations `systemActive` and `environmentInitial`, respectively, a hot violation has occurred in some requirement MSD, no hot violation has occurred in any assumption MSD, there is no active event in any assumption MSD, and no minimal delay enabled in any assumption MSD.

4.6 Compositional synthesis

The synthesis suffers greatly from the state space explosion problem [HMS08]. With every MSD, even with every lifeline that is added to an MSD specification, the complexity of the synthesis problem grows exponentially. Therefore there will always be a limit in the size of a specification for which synthesis can be carried out on the available hardware and within an acceptable time span. This section explains how in certain cases the synthesis problem can be *decomposed*, which means that the consistency of the specification can be implied from the consistency of smaller parts of the specification, which can typically be determined much faster than the consistency of the original specification.

This section first explains the principles of compositional reasoning techniques in Sect. 4.6.1, focusing on approaches based on the *assume-guarantee* paradigm. Second, a *compositional synthesis technique* for the synthesis of MSD specifications is presented in Sect. 4.6.2. Section 4.6.3 shows how this technique is successfully applied to the MSD specification of a *production cell*, an industrial production robot. Last, Sect. 4.6.4 more formally shows that the compositional synthesis technique is *sound*, i.e., that it is valid to imply the consistency of the original specification from the consistency of its parts.

4.6.1 Compositional reasoning

In formal verification, techniques for the *compositional verification* have been intensively researched. If it turns out not to be feasible to check whether a certain system satisfies a given specification, the idea is to decompose the system and the specifications such that, if the parts of the system fulfill certain local properties, it implies that the system fulfills the original specification, in the following also called the *global* specification. Since there are usually dependencies among the system parts, each part has to be verified in the context of an environment that consists of the environment of the system and the other parts of the system.

To verify the properties for one part without regarding the complete behavior of the other parts, it is necessary to abstract from the behavior of the other parts. Sometimes there exist properties that each system part can assume about the other system parts such that the verification of each single system part can be carried out more efficiently. In turn, each system part must be checked to guarantee the properties assumed by the other parts. This form of compositional reasoning is called the *assume-guarantee paradigm* (sometimes also *rely-guarantee*- or *assume-commitment paradigm*) [MC81, Jon83, Pnu85, GL94, CGP99].

There are, however, challenges in applying the assume-guarantee paradigm. An adequate decomposition of the system and its specification must be found, as well as a set of adequate assume/guarantee properties. Adequate here means, on the one hand, that the system parts should satisfy the assume/guarantee properties if the system adheres to the specification. On the other hand, the assume/guarantee properties must be sufficiently small in order to reduce the complexity of the single verification steps compared to the complexity of verifying the whole system. Often a given system is already decomposed into different modules, but, depending on the properties that shall be checked for the system, there may not always exist sufficiently small assume/guarantee properties [MWW08].

Designing an assume-guarantee reasoning technique is not trivial, because a particular application of the assume-guarantee paradigm may not be *sound*. This means that, if certain properties hold for parts of the system, and the conjunction of these properties imply the global specification, it cannot generally be implied that the composed system satisfies the global specification. For example, if liveness properties can be shown for parts of a system, but there is a cyclic dependency between these parts, this does not necessarily imply that the system is live. Imagine a system consisting of two parts and for both parts it can be shown that they progress provided that the opposite system part progresses. Both local properties imply that the whole system progresses, but it may be that the parts mutually wait for each other and therefore the composed system in fact never progresses [Sta85, Pnu85, CGP99].

The principle of compositional verification techniques can also be applied to synthesis problems: whenever it can be implied that a system satisfies its specification, provided that parts of the system satisfy their local specifications (i.e., the application of the assume/guarantee paradigm is sound), it is also valid to imply that if there exist implementations for the system parts that satisfy the local specifications, and thus the local specifications are consistent, the composition of these parts constitutes an implementation of the global specification, and thus the global specification is consistent. In the following, this kind of reasoning is called *compositional synthesis*.

By having introduced the concept of assumption MSDs within the scope of this thesis and a technique for synthesizing controllers from MSD specifications that contain assumption MSDs, an important step is taken towards being able to apply the assume-guarantee principle for the compositional synthesis of MSD specifications. Compositional synthesis techniques have not been studied for MSD or LSC specifications before. (Kugler and Segal refer to another kind of “compositional synthesis”, see a discussion on the related work in Sect. 8.2.)

4.6.2 The compositional synthesis technique

This section presents the *compositional synthesis technique* for MSD specifications. The technique consists of three steps that describe how to decompose a given global specification into two *part specifications* and how additional assume/guarantee properties can be added to these part specifications. If controllers can be successfully synthesized from the part specifications created according to the three steps described in the following, the composition of these controllers constitutes an implementation of the global specification. Accordingly, it is valid to imply the consistency of the global specification from the consistency of its part specifications.

Note, however, that the three decomposition steps require a high amount of creativity in order to find two part specifications that can be successfully synthesized. Even if the global specification is consistent, there may not always exist a decomposition of the global specification into two consistent part specifications according to the three decomposition steps.

Section 4.6.3 shows how this technique is applied to a concrete MSD specification. The compositional synthesis technique is sound, because cyclic dependencies are not allowed among the part specifications. The soundness is shown more formally in Sect. 4.6.4, based on a proof of Stark [Sta85].

Figure 4.12 illustrates the steps of the compositional synthesis technique applied to an abstract example. It shows how to construct two part specifications MS_a and MS_b from a given global specification MS . At the top, the figure shows a system consisting of objects $\underline{o1}, \dots, \underline{o7}$. From the perspective of the specification MS , the objects $\underline{o3}, \dots, \underline{o7}$ are system objects and the objects $\underline{o1}$ and $\underline{o2}$ are environment objects. The former set of objects are called O_{sys} , the latter is called O_{env} . The global specification consists of a set of assumption MSDs A and a set of requirement MSDs G (“guarantee”).

In order to decompose the synthesis problem, first, two part specifications MS_a and MS_b are created as shown in the middle of Fig. 4.12. Creating these part specification requires the two following steps:

Step 1 (Subdivide the set of system objects). Create two part specifications MS_a and MS_b from the global specification MS such that objects that are system objects in the global specification are system objects in either one of the part specifications. The set of system objects in the first part specification MS_a is called O_a , the set of system objects in the second part specification MS_b is called O_b . The objects O_{env} and O_b are environment objects in MS_a ; The objects O_{env} and O_a are environment objects in MS_b .

Step 2 (Create subsets of MSDs for the part specifications). Divide the set of requirement MSDs in the global specification among the part specifications MS_a and MS_b such that, first, the union of the sets of requirement MSDs in the part specifications is equal to the set of requirement MSDs in the global specification, i.e., each requirement MSD in MS must appear in MS_a or MS_b . Second, each part specification may contain any subset of the assumption MSDs in the global specification.

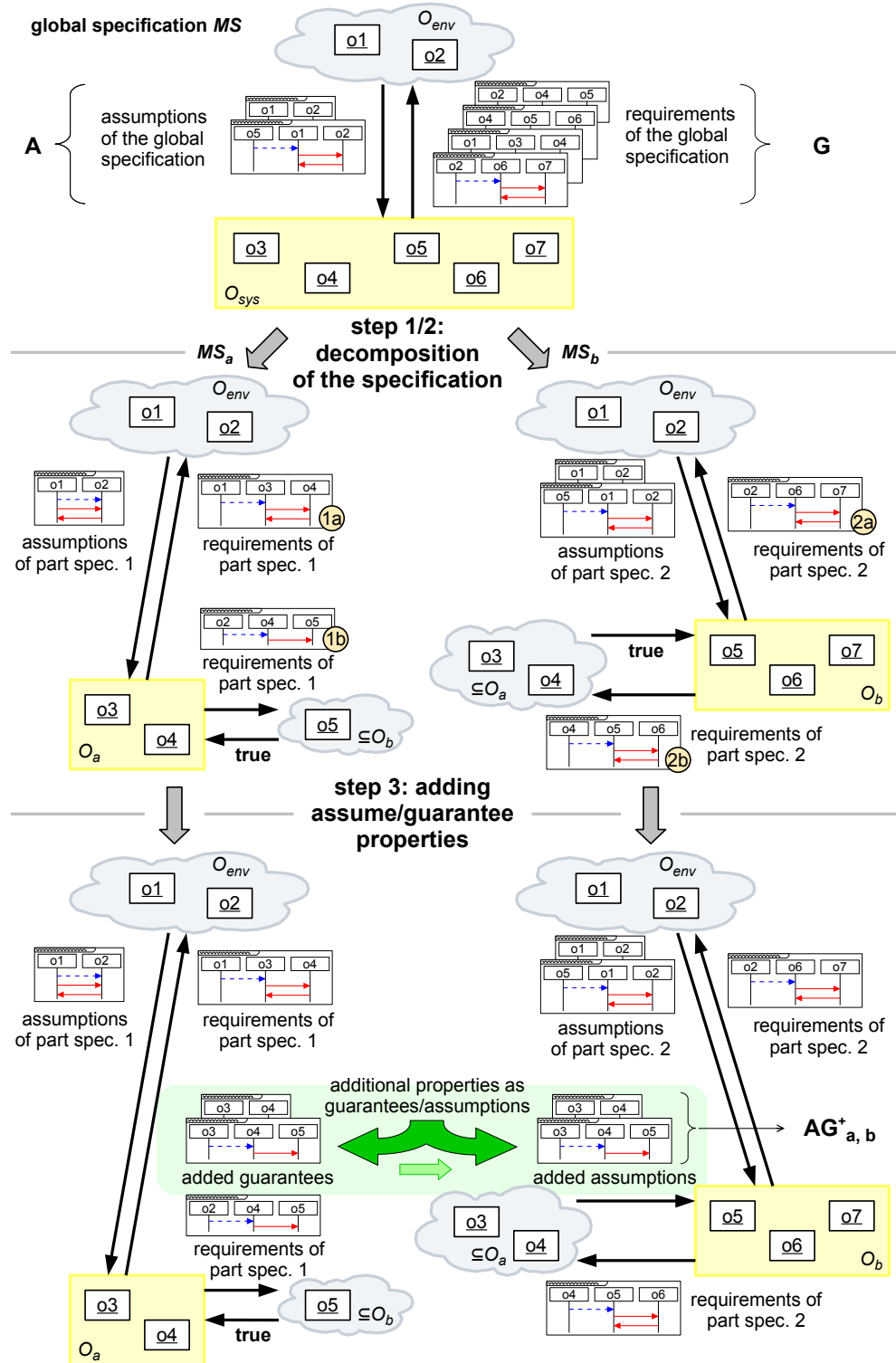


Figure 4.12: A sketch of the compositional synthesis approach

The middle section of Fig. 4.12, illustrates how the global specification of the abstract example is decomposed into two part specifications MS_a and MS_b according to Step 1 and Step 2. It shows that the objects $\underline{o3}$ and $\underline{o4}$ are the system objects in MS_a, O_a , and the objects $\underline{o5}, \underline{o6}$ and $\underline{o7}$ are the system objects in MS_b, O_b . The system objects in MS_a, O_a , are environment objects in MS_b . Sometimes the lifelines of the MSDs in the part specifications only refer to a subset of their environment objects. The middle left of Fig. 4.12 for example shows that MS_a only refers to a subset of O_b , namely the object $\underline{o5}$.

In the middle section of Fig. 4.12, the MSDs of the global specification are split up according to Step 2. The message names are not shown in this abstract example. The purpose of Fig. 4.12 is to indicate that typically the MSDs are split up into four different groups. There are such requirement MSDs in MS_a that describe how the objects O_a are required to interact with the objects O_{env} (1a) resp. with the objects O_b (1b). Likewise, there are such requirement MSDs in MS_b that describe how the objects O_b are required to interact with the objects O_{env} (2a) resp. with the objects O_a (2b). (1a and 1b resp. 2a and 2b may not necessarily be disjoint, i.e., there may be MSDs in MS_a that refer to both objects from env and b , and there may be MSDs in MS_b that refer to both objects from env and a . Note also that the MSDs in these sets not only specify how the system objects interact with their environment objects, but also how they interact with each other.)

If already at this point, after performing Steps 1 and 2, both part specifications can be shown to be consistent, this implies that there exist controllers a and b for the objects O_a and O_b that fulfill the specification MS_a resp. MS_b for any implementation of the environment objects O_{env} that satisfies the environment assumptions and for any implementation of the O_b resp. O_a . In other words, there exists a controller a of the objects O_a that can satisfy MS_a regardless of the behavior of the objects O_b and there exists a controller b of the objects O_b that can satisfy MS_b regardless of the behavior of the objects O_a . This is indicated in the middle section of Fig. 4.12 by the arrows labeled “true”: In MS_a , no assumptions are made about the behavior of O_b . Likewise, there are no assumptions made about the behavior of O_a in MS_b . This means that the composition of the controllers for a and b is an implementation of MS and, thus, MS is consistent. (In the following, for simplicity, instead of saying “controller for O_a ” or “controller for O_b ”, we simple say “ a ” or “ b ”, for example “there exists an a satisfying MS_a ” means “there exists a controller for O_a satisfying MS_a ”.)

For most specifications it will however not be possible to easily decompose a global specification into two independently consistent parts, by only applying Step 1 and 2. Very often, it will be necessary to furthermore assume that a will not do arbitrarily “bad” things to force b to violate the specification MS_b . This can be done by adding adequate assumption MSDs to MS_b . But, whenever an assumption about a is added to MS_b , it must be in turn ensured that there exists an a which guarantees these properties. This can be done by taking the same MSD that was added as an assumption MSD to MS_b and adding it as a requirement MSD to MS_a . The additional assumptions that a guarantees towards b , and, consequently, b assumes about a , are called $AG_{a,b}^+$.

Sometimes, we would also like to add assume/guarantee properties in the other direction: it must be assumed that b will not do arbitrarily “bad” things to force a to violate the specification MS_a . However, if there are assume/guarantee properties added in both directions, it may happen that both a and b can discharge their obligation to fulfill their requirements by violating the assumptions they make about their opposites. This way, a and b may be able to fulfill their part specifications simply by not fulfilling the properties that their opposites assume about them. This means that, even if the global assumptions A hold, the local assumptions may not hold. Thus, the components a and b need not fulfill any requirements and, consequently, the components need not fulfill the global requirements G . It follows that the composition of a and b may not constitute a valid implementation of the global specification, and it is therefore not valid to imply the consistency of the global specification from the consistency of the part specifications. Therefore, the compositional synthesis technique presented here does only allow additional assume/guarantee properties to be added in one direction.

As an example, imagine a system of two robot arms where we require that they pass plates back and forth between each other infinitely often. Now we could specify locally for each arm that it must pass a plate to the other arm, assuming that the other arm passes a plate back. But now it could be that neither of the two arms passes any plates to the other. This way, each arm could satisfy its local specification, since the local assumptions are violated. However, the system would not satisfy the original specification.

Step 3 (Add assume/guarantee properties to the part specifications). Assumption MSDs may be added to MS_b provided that for each assumption MSD added to the MS_b an identical MSD appears as a requirement MSD in MS_a .

The lower part of Fig. 4.12 illustrates this process of adding MSDs $AG_{a,b}^+$ as additional assumptions/guarantees to the specifications MS_b resp. MS_a .

For a decomposition of the specification, it may sometimes furthermore be desirable or necessary to *decompose a system object into two objects*, where then one object can fulfill the functions specified in one part specification and the other can fulfill the functions specified by the other part specification. When doing so, the MSD specification has to be modified accordingly. The goal is that the modified specification can then be split up in such a way that one part of the specification specifies the behavior of the first object and another part of the specification specifies the behavior of the second object; optimally, and in order to successfully apply the described compositional synthesis technique, the behavior of the first object is independent from the behavior of the second and only a few MSDs remain that specify what the second object must do in reaction to the behavior of the first (an example will follow shortly).

If a system object is decomposed into two objects, then the MSD specification can always be changed such that for the two system objects resulting from the decomposition of the initial system object, the same externally visible behavior is specified as for the initial system object. Decomposing a system object

and changing the MSD specification accordingly must be done before Step 1. Therefore, this step is called “Step 0”, and works as follows.

Step 0 (Decomposing system objects). A system object in the specification may be decomposed into two system objects. Then in each MSD in the specification, each lifeline that represents the initial object must be split into two lifelines that represent the two objects resulting from the decomposition of the initial system objects. Then the set of messages that the initial object sent and received can be split up into the messages that the first object resulting from the decomposition sends and receives and the messages that the second object sends and receives. In some MSDs, the effect may be that one of the lifelines does not send or receive any messages. This lifeline can then be removed from the MSD. In the MSDs where no lifeline can be removed, the lifelines must be synchronized so that an ordering of the events as on the initial lifeline is preserved. This can be achieved by introducing (hot or cold) conditions with the expression `true` that cover both lifelines. These conditions must always be introduced between two events (i.e., the sending or receiving of a message) where one event on one lifeline is followed by an event on the other lifeline. The resulting externally visible behavior that the specification specifies for the two objects then is the same as the behavior specified for the initial object.

Note that the steps described above may require a high creativity in finding two good part specifications. Two part specifications are “good” if they can be successfully synthesized when the global specifications is consistent. The part specifications should in particular not be bigger than the global specification (in terms of the number of MSDs), because then it is likely that the synthesis of controllers for the part specifications takes longer than the synthesis of a controller for the global specification. Note that “good” result after Step 3 rely on a good decomposition of the requirement MSDs in Step 2, which relies on an adequate decomposition of the object system in Step 1. Also note that it may be possible to decompose a specification multiple times by again applying Steps 0-3 to the part specifications.

In the following section, the compositional synthesis approach is further illustrated by an example. A more formal argument on the soundness of the compositional synthesis technique consisting of the three steps described above is given in Sect. 4.6.4.

4.6.3 Example: the compositional synthesis of the production cell specification

This section presents the compositional synthesis of the *production cell* example. The production cell is an industrial production system, which processes metal blanks into plates. It was considered as a case study within the KORSo project [LL94, LL06, BJ95]. Here a simplified version of the production cell is considered.

The production cell example

A sketch of the production cell is shown in Fig. 4.13. It consists mainly of two transport belts, a press, and two robot arms. Metal blanks are transported to the production cell on the feed belt within certain time intervals. At the end of the feed belt, these blanks arrive on a table where they have to be picked up by the first robot arm. This robot arm transports the blank to a press, where it is processed to a plate. Then a second robot arm picks up the plate and transports it to the deposit belt. It is required that a blank is picked up from the table before the next blank arrives. Also, the second robot arm must pick up the pressed plate before the first robot arm can place a new blank into the press. The controller of the production cell can control the press and the movement of the robot arms. The arrival of the blanks, the time for pressing the blanks to plates, and the time that the robot arms require for moving between the table and the press resp. the press and the deposit belt are uncontrollable, but certain minimal and maximal delays can be assumed.

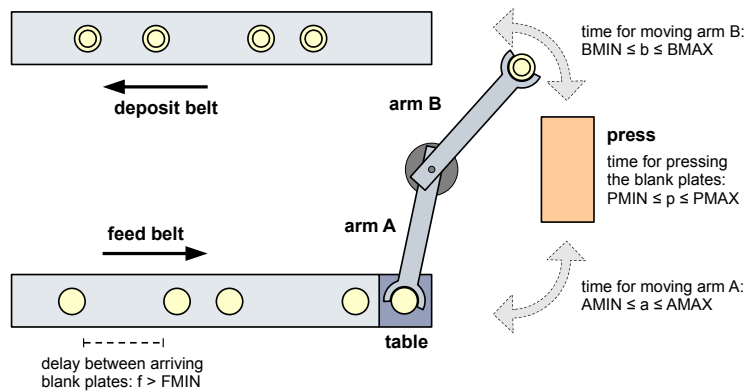


Figure 4.13: A sketch of the production cell

Depending on the assumed delays, the specification may be inconsistent. If the intervals in which blanks arrive at the press are too short, it may be that the robot arms are not fast enough to transport the blanks resp. plates. It may also be that the press takes too much time, so that the first robot arm cannot release the next plate into the press quickly enough.

Table C.1 in the Appendix (Sect. C.3, p. 259) lists the requirements of the (single) use case of the production cell. Section. C.3.2 presents the corresponding MSD specification (the requirement MSDs are shown in Fig. C.26, the assumption MSDs are shown in Fig. C.27).

It turns out that, due to the interdependent time constraints, the synthesis for the specification is hardly feasible. Synthesizing an admissible play-out strategy satisfying the *AG* winning condition from a consistent specification (i.e., values for the assumed time intervals for which the specification is realizable) took 34 minutes. Synthesizing a controller strategy satisfying the *AGAF* winning condition was not possible at all; the memory limit of 4GB was exceeded after 7 hours (see Sect. C.3). The idea is therefore to split the specification of the production cell in two parts. Controllers for the two part specifications could be

synthesized much faster than a controller for the global specification. Synthesis with the *AG* winning condition takes 0,01 + 0,5 minutes, synthesis with the *AGAF* winning condition was now possible and took 0,09 + 2,03 minutes (see row one in Fig. C.28 and C.32).

The decomposition of the production cell specification

The specification of the production cell is decomposed as follows. The first part describes how the first robot arm transports the blanks to the press, and the second part describes how the press processes the blanks that arrive at the press as well as how the second robot arm transports the pressed plates to the deposit belt. With this decomposition, the second part depends on the first part. Whenever the first arm releases a blank into the press, the press must process the plates and then arm B must transport the plates to the deposit belt. In particular, it must be assumed that the first arm does not place blanks into the press too quickly, because otherwise blanks may be placed into the press before the press is finished or before arm B picks up the pressed plate from the press. Therefore, it is necessary to additionally assume that the first arm will release blanks into the press only within certain minimal time intervals. If such an assumption is made by the second part, this property must consequently be guaranteed by the first part.

Section C.3.3 illustrates this decomposition of the specification into two part specifications. The MSD `BlankArrivalAtPressDelay` occurs as an assumption in the second part specification (see Fig. C.31) and as a requirement in the first part specification (see Fig. C.30). It is described in the following how these two part specifications are created.

Step 0 (Decomposing system objects): One particular issue with this example is that the object system of the original specification just consists of one system object (`c:Controller`), which receives the signals from the table sensor and controls arm A, the press and arm B. The top of Fig. 4.14 shows this controller object `c:Controller` and the environment objects `ts:TableSensor`, `a:ArmA`, `p:Press` and `b:ArmB` that the controller interacts with. The lines represent where objects interchange messages with each other. If there is just one system object, the specification cannot be decomposed in a meaningful way. Therefore, as described in Step 0 in Sect. 4.6.2, the global specification is changed such that `c:Controller` is replaced by two system objects `c1:Controller1` and `c2:Controller2`, which control arm A resp. the press and arm B. The decomposition of the controller is shown in the middle of Fig. 4.14.

According to Step 0, this requires the MSDs in the specification to be changed, too. The set of messages sent and received by `c:Controller` is split up so that `c1:Controller1` only interacts with `ts:TableSensor` and `a:ArmA`. `c2:Controller2` only interacts with `p:Press` and `b:ArmB`. Many MSDs in the specification will now only refer to either `c1:Controller1` or `c2:Controller2`. Figure 4.15 shows how the MSD `PressPlateAfterArmAReleasesBlankPlate`, where messages of both `c1:Controller1` and `c2:Controller2` occur, is changed according to Step 0.

After decomposing the `c:Controller` object, we can split up the production cell specification into two part specifications.

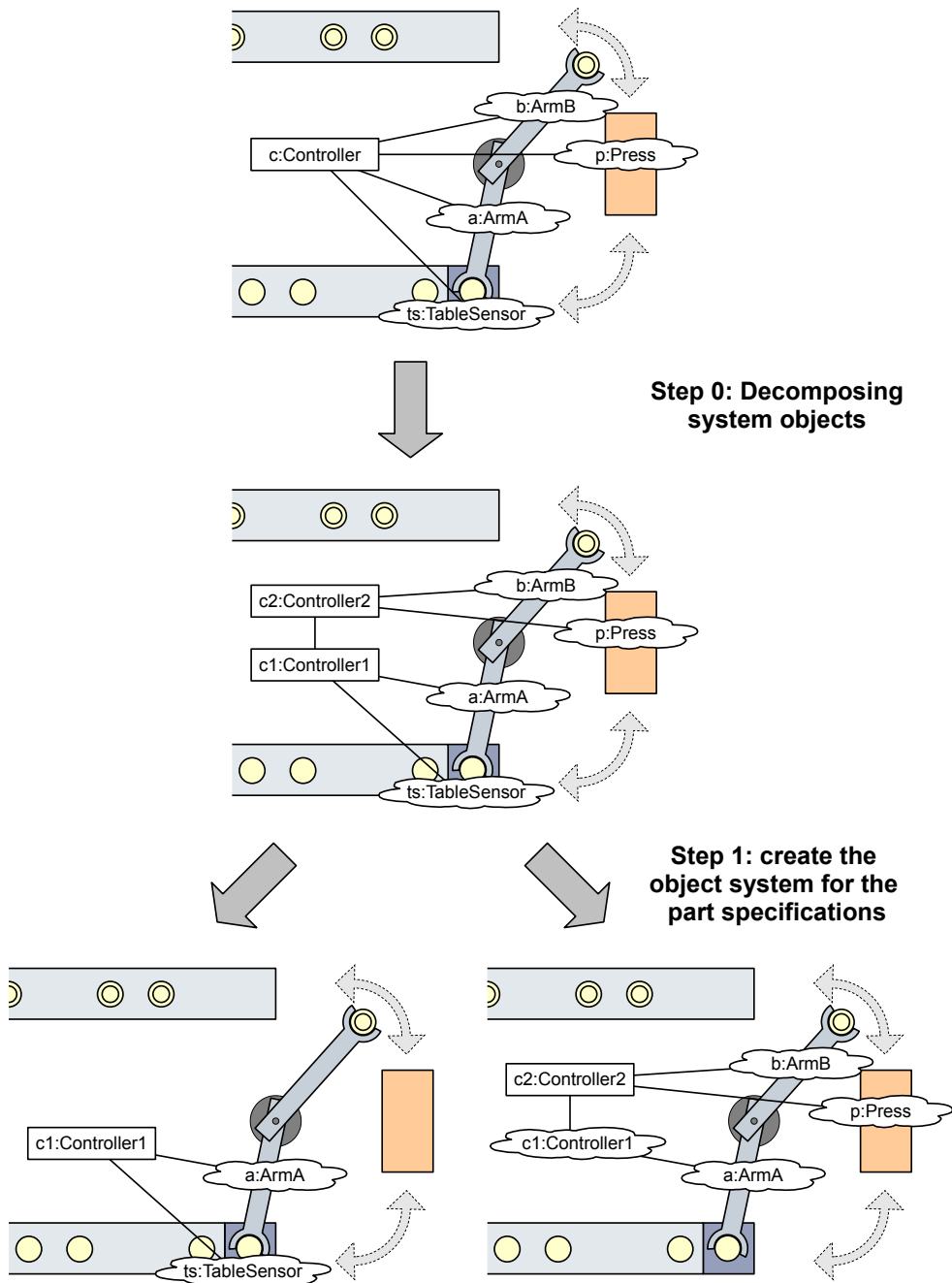


Figure 4.14: Step 0 and Step 1 in the decomposition of the production cell specification

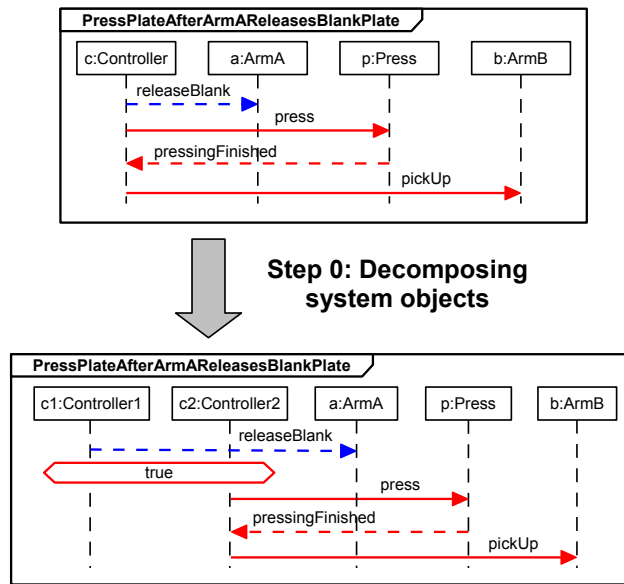


Figure 4.15: Splitting the lifeline of MSD `PressPlateAfterArmAReleasesBlankPlate` according to Step 0

Step 1 (Create the object systems for the part specifications): With the system object `c:Controller` replaced by the objects `c1:Controller1` and `c2:Controller2`, two part specifications can be created (according to Step 1, see Sect. 4.6.2 above) from the global specification with object systems where `c1:Controller1` appears as a system object in the first part specification and `c2:Controller2` appears as a system object in the second part specification. The bottom of Fig. 4.14 shows the object systems that are created for the two part specifications. (See also the collaboration diagrams in Fig. C.30 and Fig. C.31). The connector lines between the objects at the bottom of Fig. 4.14 again describe where two objects interchange messages. The connector lines are implied by the subsets of MSDs that appear in the part specifications, as explained in the following.

Step 2 (Create subsets of MSDs for the part specifications): Subsets of the MSDs from the global specification are taken to form the MSDs of the first and second part specification. This example allows us to easily decompose the requirement MSDs among the part specifications, while adhering to the constraints formulated for Step 2 above (see Sect. 4.6.2): All the requirement MSDs with lifelines referring to just the first controller are members of the first part specification. All remaining requirement MSDs are placed in the second part specification. The assumption MSDs are subdivided among the part specifications by the same principle. (This also explains which environment objects from the global specification must be represented in the part specification collaboration diagrams: these are all such objects which occur in the MSDs that are placed in the respective part specification.)

After this decomposition, the first part specification can be shown to be consistent, provided that the time intervals in which the blanks arrive are not too short for the first robot arm to pick them up. It will however not be pos-

sible to show that the second part specification is consistent. Synthesis will show that the requirements are always easily violated by `c1:Controller1` sending the `releaseBlank` message to the first arm without any delay. Therefore, an additional property must be added as an assumption to the second part specification and, accordingly, as an additional requirement (guarantee) to the first part specification.

Step 3 (Add assume/guarantee properties to the part specifications): An assumption is added to the second part specification that specifies a minimal delay between occurrences of the `releaseBlank` event (MSD `PressPlateAfter-ArmARelasesBlankPlate` (2), see Fig. C.31). Accordingly, a copy of this MSD is added as a requirement MSD in the first part specification (see Fig. C.30). The tables in Fig. C.32 and C.33 present the results from synthesizing controllers from the part specifications with different values for the timing assumptions made about the blank arrival, the arms' movement, and the pressing process.

4.6.4 The compositional synthesis technique is sound

This section shows more formally that if controllers could be successfully synthesized from the part specifications MS_a and MS_b constructed according to the steps described in Sect. 4.6.2, the composition of these controllers constitutes an implementation of the global specification, i.e., there exists an implementation for the global specification, and it is therefore valid to imply the consistency of the global specification from the consistency of the part specifications. A proof sketch is presented, inspired by the compositional proof technique presented by Stark [Sta85].

Before sketching the proof, a number of conditions are formulated that follow for the part specifications MS_a and MS_b when constructed according to the steps above. Furthermore, some notations are defined that will be used in the following. Note that the proof sketch abstracts from many details, such as the exact semantics of the MSDs that results from their encoding in the TGAs of UPPAAL TIGA. Furthermore, a thorough proof would of course also require to formally show the correctness of the synthesis technique, which is beyond the scope of this thesis.

Definitions

In the following, we consider specifications of a system of objects O . The objects O can interchange messages. An infinite sequence of message interchanges is called a *run* of the system, which we denote as π . The objects O can be controlled by a *controller*, which defines a particular *set of runs* for the system. For a controller c , the set of runs is denoted as $L(c)$, also called the *trace language* of the system controlled by the controller c . A controller for the objects O can consist of the *parallel composition* of multiple controllers for disjoint subsets of O . We write for example $c = c1||c2||c3$ to denote that c consists of the parallel composition of $c1$, $c2$ and $c3$.

A *specification* can consist of a set of universal MSDs MS with lifelines representing objects in O . In Sect. 3.1.6, it was already explained that a universal

MSD can be described by a Büchi automaton that accepts a *set of runs* of a system. According to Harel and Maoz [HM08], we denote this set of runs of a universal MSD D as $L(D)$, also called the *trace language* of a universal MSD or the runs *accepted* by D . A run of the system *satisfies* the specification MS , denoted as $\pi \models MS$, iff the run is accepted by every MSD in MS .

$$\pi \models MS \equiv \pi \in L(D_i), \forall D_i \in MS \quad (4.5)$$

A specification can also be formed by the *union* of two sets of MSDs MS_1 and MS_2 , denoted as $MS_1 \cup MS_2$. From (4.5) follows that

$$\pi \models MS_1 \cup MS_2 \equiv \pi \models MS_1 \wedge \pi \models MS_2 \quad (4.6)$$

From (4.5) also follows that if a set of MSDs MS_1 is a subset of another set of MSDs MS_2 and a run satisfies MS_2 , the run also satisfies MS_1 .

$$(MS_1 \subseteq MS_2 \wedge \pi \models MS_2) \Rightarrow \pi \models MS_1 \quad (4.7)$$

A specification MS can furthermore be formed from two sets of MSDs MS_1 and MS_2 by writing $MS_1 \Rightarrow MS_2$. $\pi \models MS_1 \Rightarrow MS_2$ means that if π satisfied MS_1 , it also satisfies MS_2 .

$$\pi \models MS_1 \Rightarrow MS_2 \equiv \pi \models MS_1 \Rightarrow \pi \models MS_2 \quad (4.8)$$

(The MSDs MS_1 and MS_2 form a specification where MS_1 are the *assumption MSDs* and MS_2 are the *requirement MSDs*.)

A controller c of the objects O *satisfies* a specification MS iff every run of c satisfies MS .

$$c \models MS \equiv \pi \models MS, \forall \pi \in L(c) \quad (4.9)$$

A specification subdivides the set of objects O into (controllable) *system objects* O_{sys} and (uncontrollable) *environment objects* O_{env} ($O = O_{sys} \cup O_{env}$, $O_{sys} \cap O_{env} = \emptyset$). An MSD specification MS is *consistent* iff there exists a controller s for the system objects such that for every possible controller e of the environment objects, the controller formed by the parallel composition of s and e satisfies the specification (see also Sect. 3.1.9).

$$MS \text{ is consistent} \equiv \exists s \forall e : e || s \models MS \quad (4.10)$$

Such a controller s is also called the *witness* for the consistency of MS .

The two part specifications

Let MS be a global specification consisting of a set of assumption MSDs A and the requirement MSDs G . Then applying the Steps 1-3 (see Sect. 4.6.2) yields two part specifications MS_a and MS_b with the assumption MSDs A_a resp. A_b and the requirement MSDs G_a resp. G_b . Using the notation defined above, we write $MS = A \Rightarrow G$, $MS_a = A_a \Rightarrow G_a$ and $MS_b = A_b \Rightarrow G_b$. The specifications refer to a system of objects O that can be subdivided into the disjoint subsets O_{env} , O_a and O_b such that $O_a \cup O_b$ and O_{env} are the system resp. environment

objects of MS , O_a and $O_{env} \cup O_b$ are the system resp. environment objects of MS_a , and O_b and $O_{env} \cup O_a$ are the system resp. environment objects of MS_b . (As described above, it may be that the MSDs in the part specifications MS_a and MS_b only refer to a subset of the $O_{env} \cup O_b$ resp. $O_{env} \cup O_a$, but this has no further implications for the following reasoning.)

In Step 2, the MSDs in MS are subdivided into several, not necessarily disjoint subsets that later recur in the part specifications MS_a and MS_b . In Step 3, further MSDs can be added to G_a and A_b . These subsets were illustrated in Fig. 4.12; Fig. 4.16 again shows part of this diagram and introduces the following abbreviations for these subsets of MSDs:

- $AG_{env,a}$: the MSDs that specify the requirements of O_{env} towards O_a
- $AG_{env,b}$: the MSDs that specify the requirements of O_{env} towards O_b
- $AG_{a,env}$: the MSDs that specify the requirements of O_a towards O_{env}
- $AG_{b,env}$: the MSDs that specify the requirements of O_b towards O_{env}
- $AG_{a,b}$: the MSDs that specify the requirements of O_a towards O_b
- $AG_{b,a}$: the MSDs that specify the requirements of O_b towards O_a
- $AG_{a,b}^+$: the additional MSDs that are added as requirement MSDs to MS_a and as assumption MSDs in MS_b

According to Step 2-3, the following conditions hold for the above subsets of MSDs and A , G , A_a , G_a , A_b , and G_b .

The additional assumptions/requirements MSDs added in Step 3, $AG_{a,b}^+$ are subset of $AG_{a,b}$.

$$AG_{a,b}^+ \subseteq AG_{a,b} \quad (4.11)$$

G consists of the MSDs that specify requirements of O_a and O_b towards O_{env} , the requirements of O_b towards O_a , and the requirements of O_a towards O_b . Not part of G are the MSDs that are added as additional assumptions/requirements to the part specifications later on, $AG_{a,b}^+$. We therefore write

$$G \subseteq AG_{a,env} \cup AG_{a,b} \cup AG_{b,env} \cup AG_{b,a} \quad (4.12)$$

The MSDs A are subdivided into the sets of MSDs that specify the requirements of O_{env} towards O_a and the MSDs that specify the requirements of O_{env} towards O_b .

$$A = AG_{env,a} \cup AG_{env,b} \quad (4.13)$$

G_a consists of the MSDs which express the requirements of the objects O_a towards the environment objects O_{env} and the requirements of the objects O_a towards the objects O_b (including the additional MSDs added in Step 3, $AG_{a,b}^+$).

$$G_a = AG_{a,env} \cup AG_{a,b} \quad (4.14)$$

Accordingly, G_b consists of the MSDs which express the requirements of the objects O_b towards the environment objects O_{env} and the requirements of the objects O_b towards the objects O_a .

$$G_b = AG_{b,env} \cup AG_{b,a} \quad (4.15)$$

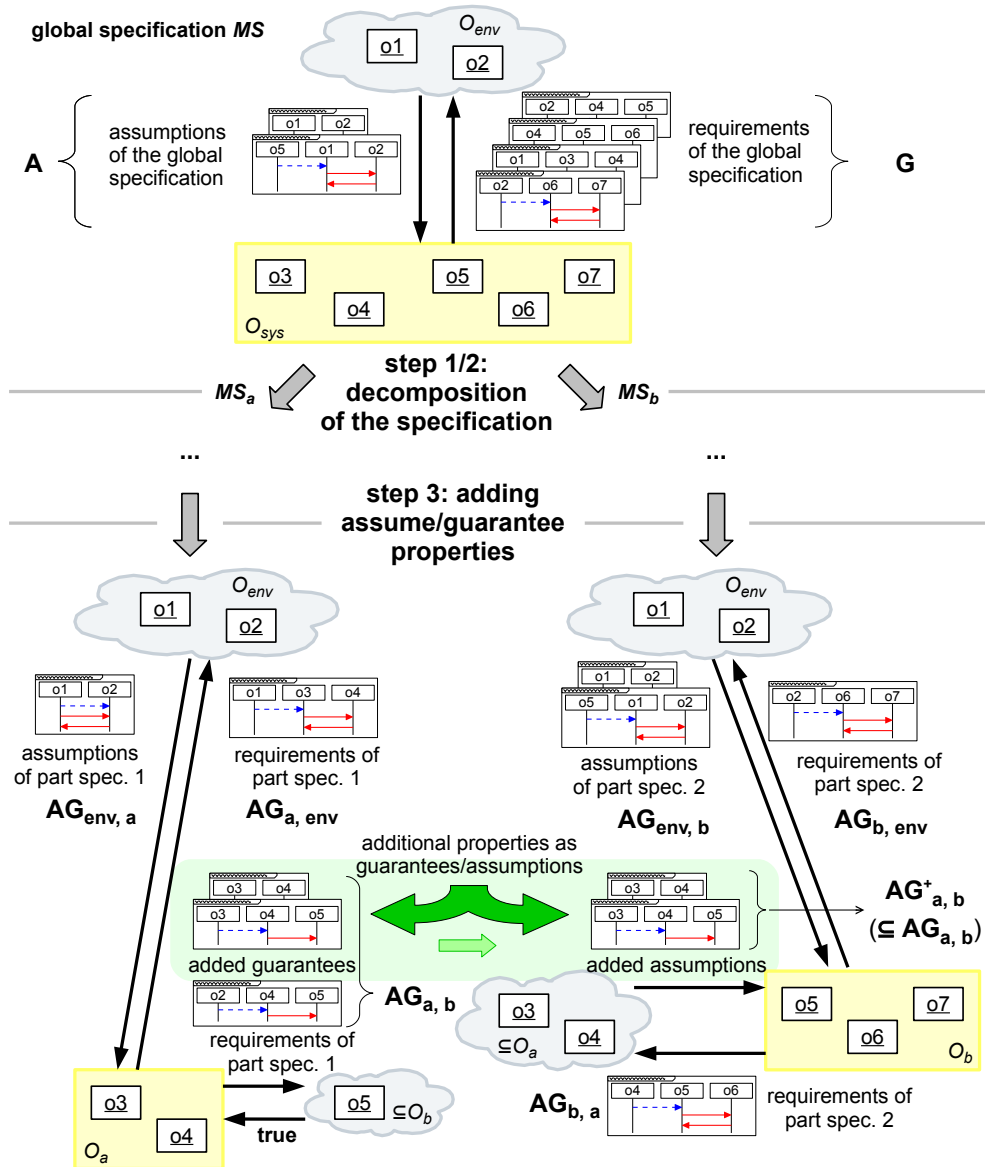


Figure 4.16: A sketch of how the compositional synthesis approach is mapped to the proof technique of Stark

A_a consists of the MSDs which specify the requirements of objects O_{env} towards the objects O_a .

$$A_a = AG_{env,a} \quad (4.16)$$

A_b consists of the MSDs which specify the requirements of objects O_{env} towards the objects O_b and the requirements of the objects O_a towards the objects O_b (including the additional MSDs added in Step 3, $AG_{a,b}^+$).

$$A_b = AG_{env,b} \cup AG_{a,b} \quad (4.17)$$

Proof

We can now prove that if the synthesis of controllers for MS_a and MS_b shows that MS_a and MS_b are consistent, then also MS is consistent. The proof works by contradiction.

Proof (MS is consistent if MS_a and MS_b are consistent). We assume that MS_a and MS_b are consistent, but MS is inconsistent. If MS_a is consistent, this means that there exists a controller a for the objects O_a such that the parallel composition of a and any controllers e and b' for the objects O_{env} resp. O_b satisfies MS_a . If MS_b is consistent, this means that there exists a controller b for the objects O_b such that the parallel composition of b and any controllers e and a' for the objects O_{env} resp. O_a satisfies MS_b .

$$\exists a, b : e||a||b' \models A_a \Rightarrow G_a \wedge e||a'||b \models A_b \Rightarrow G_b, \forall e, a', b' \quad (4.18)$$

If MS is inconsistent, this means that there do not exist any controllers a' and b' for the objects O_a and O_b such that the parallel composition of a' and b' with any controller e for the environment objects O_{env} satisfies MS .

$$\nexists a', b' : e||a'||b' \models A \Rightarrow G, \forall e \quad (4.19)$$

In particular, if we choose a' and b' to be the witnesses a and b for the consistency MS_a and MS_b , $e||a||b$ would not satisfy MS , e being any possible controller for the objects O_{env} in the following.

$$e||a||b \not\models A \Rightarrow G, \forall e \quad (4.20)$$

$e||a||b \not\models A \Rightarrow G$ means that there exists a run π in $L(e||a||b)$ such that $\pi \models A$, but $\pi \not\models G$. According to (4.12) and (4.7), this means that

$$\pi \not\models AG_{a,env} \cup AG_{a,b} \cup AG_{b,env} \cup AG_{b,a} \quad (4.21)$$

which, because of (4.6), can be rewritten as

$$\pi \not\models (AG_{a,env} \cup AG_{a,b}) \vee \pi \not\models (AG_{b,env} \cup AG_{b,a}) \quad (4.22)$$

or, because of of properties (4.14) and (4.15), as

$$\pi \not\models G_a \vee \pi \not\models G_b \quad (4.23)$$

This means that the requirements of either one of the part specifications is violated. But because a and b satisfy MS_a and MS_b for every possible controller e of O_{env} (4.18), this again implies that the assumptions of at least one part specification is violated.

$$\pi \neq A_a \vee \pi \neq A_b \quad (4.24)$$

which, due to properties (4.16) and (4.17) can be rewritten as

$$(\pi \neq AG_{env,a}) \vee (\pi \neq (AG_{env,b} \cup AG_{a,b}^+)) \quad (4.25)$$

which, due to (4.6), is equivalent to

$$\pi \neq AG_{env,a} \vee \pi \neq AG_{env,b} \vee \pi \neq AG_{a,b}^+ \quad (4.26)$$

Because the assumption of the proof states that $\pi \models A$, and because of property (4.13), we can imply that $\pi \models AG_{env,a}$ and $\pi \models AG_{env,b}$. This allows us to further simplify the above condition to

$$\pi \neq AG_{a,b}^+ \quad (4.27)$$

But because $\pi \models AG_{env,a}$, and because we have assumed that a satisfies MS_a for any controller of O_{env} and O_b , $e \parallel a \parallel b' \models AG_{env,a} \Rightarrow (AG_{a,env} \cup AG_{a,b})$, $\forall e, b'$ (see (4.16), (4.18) and (4.14)), π must satisfy $AG_{a,b}$. Because of (4.11) and (4.7), it cannot be the case that $\pi \neq AG_{a,b}^+$.

This contradicts the assumption of the proof and therefore proves that if MS_a and MS_b if constructed from MS according to the above guidelines are consistent, also MS is consistent. \square

4.7 Different kinds of consistency

This section discusses how the encoding of MSD specifications as a TGA network as presented in Sect. 4.3 and 4.5 can be modified slightly in order to check different kinds of consistency.

4.7.1 Disallowing to delay steps in the timed setting

One change in the encoding of the TGA system in a timed setting that can drastically increase the efficiency of the synthesis is to disallow the system to delay the execution of active system events. This restriction is also made in the timed play-out, where it is required that active events are executed immediately [HM02b]. This can be achieved by adding a guard `not isActive(sys)` or `hotViolation` or `hotEnvViolation` to the edge in the system automaton that leads from the location `systemActive` to the location `systemInactive`, like in the system automaton in the untimed case. Now the system automaton can only move from the location `systemActive` to the location `systemInactive` if there are no events active in any requirement MSD. Because `systemActive` is a committed location, it will have to produce events immediately. This greatly reduces the state space of the synthesis problem. However, then it is not possible to successfully synthesize controllers for a range of consistent timed MSD specifications.

An admissible controller can for example not be found for the consistent Drive onto merging switch use case, because the reply `enterAllowed` of the track section control must then be sent immediately (as explained in Sect. 4.5.1). Also, there will be no controller for the technical example presented in Sect. C.4.3.

However, for the production cell example, the synthesis speed can be increased from 45 minutes to *two seconds* when delaying the system steps is disallowed and using the AG winning condition. When disallowing the delay of system steps, the synthesis with the AGAF winning condition is now possible in *five seconds*. When the delay of system steps is allowed, it exceeded the memory limits of four GB of UPPAAL TIGA after seven hours. See the tables in Fig. C.28 and C.29 for the comparison of the run-times of the synthesis with different parameters for the delays in the production cell example.

4.7.2 Consistency vs. consistent executability

By the encoding of the TGA network presented in Sect. 4.3 and 4.5, the synthesis determines whether an MSD specification is *consistently executable*, which means that we check whether there is an admissible controller for the system that can satisfy the requirements by always immediately executing only such events that are currently active. Only in the timed setting, as discussed in the previous section, we additionally allow the system to delay system steps. In order to check the *consistency* of an MSD specification, we have to *modify* the encoding of the TGA network, so that the system can, first, also execute events that are not currently active, and second, that it can interrupt a super-step and wait for environment events to occur, even if there are still active events.

These modifications only concern the encoding of the system automaton. They are as follows:

- **untimed setting:**
 - **execute all events:** the guards `active(<event>)` must be removed from the edges that lead from the location `systemActive` to the location `produceEvent` in the system automaton.
 - **interrupt super-step:** The following modifications must be made:
 - * The guard `not isActive(sys)` must be removed from the edge leading from the location `systemActive` to the location `systemInactive`.
 - * The Boolean variable `block` must be declared in the system automaton template
 - * An edge from the location `systemInactive` to the location `systemActive` must be added with the guard `not block` and the update label `block=true`.
 - * The update label `block=false` must be added to the edge leading from the location `produceEvent` to the location `handleHiddenEvent`.
- **timed setting:**
 - **execute all events:** the guards `active(<event>)` must be removed from the edges that lead from the location `systemActive` to the loca-

- tion `produceEvent` in the system automaton (same modification as in the untimed setting).
- **interrupt super-step:** is already possible with the encoding presented in Sect. 4.5, see the discussion in Sect. 4.7.1.

4.8 Summary and Outlook

This chapter presented a technique for checking the consistency or consistent executability of untimed or timed MSD specifications. The technique is based on encoding the problem as a two-player game that can be solved by UPPAAL TIGA. When checking the consistency or consistent executability, UPPAAL TIGA is able to synthesize an admissible global controller or an admissible play-out strategy from the MSD specification if it is consistent or consistently executable, respectively. If the MSD specification is inconsistent or not consistently executable, UPPAAL TIGA can even generate a counter-strategy that shows how the environment can always violate the specification. The correctness of the technique was not proven formally, but tests with different consistent and inconsistent examples (see Sect. C.3 and C.4) produced the expected results, providing evidence for the correctness of the technique. Once the TGA system is created, UPPAAL TIGA takes less than a second to synthesize controllers from the example MSD specifications provided in this chapter. Sect. 7.4.2 will summarize the performance evaluation results in more detail.

Novel in this synthesis technique is in particular that environment assumptions in the form of assumption MSDs can be considered during the synthesis. Furthermore, this chapter presented a novel compositional synthesis technique.

Some issues, however, remain that will have to be addressed in the future.

4.8.1 Inconsistent environment assumptions

If the environment assumptions are overly optimistic, the specification may not capture critical situations in the system's environment. Even worse, if environment assumptions are captured in a scenario-based way, it may happen that the environment is not able to always satisfy the environment assumptions. Then we say that the environment assumptions are *inconsistent*. That means that the system is always able to satisfy its requirements, because it can force the environment to violate the environment assumptions.

Such environment assumptions are typically erroneous descriptions of the environment behavior, and it is desirable to automatically check the environment assumptions for such inconsistencies in the future.

4.8.2 Partial observability

One aspect that is not considered in this synthesis is that the system does usually not have the complete information about the state of the environment. Usually a system can only observe certain events in the environment, via sensors or external messages that it receives. Many events in the environment may therefore remain hidden to the system. This is called *partial observability*. There

are some specifications where the system cannot satisfy the requirements only because it does not see certain moves of the environment. This problem does not occur within the example MSD specifications shown in this thesis, but it may be an issue in other practical examples. In the future, the synthesis therefore has to be extended to support partial observability. For example the lifeline variables of the assumption MSDs, the diagrams variables, or events that are sent between environment objects must be obscured from the system. UPPAAL TIGA supports partial observability [Cas07]. Applying these concepts to the MSD synthesis technique, however, is future work.

Symbiosis of Simulation and Synthesis

This chapter describes an improved play-out algorithm that is capable of avoiding avoidable violations during the play-out of an MSD specification by incorporating controllers that could be successfully synthesized from parts of the specification. In the following, Sect. 5.2 explains how the play-out can be guided by controllers that could be successfully synthesized from single use case specifications. Section 5.3 explains how the play-out can be guided by controllers that could be successfully synthesized from a combination of multiple use cases. But first, Sect. 5.1 overviews the scenario-based design process that shall be supported by the techniques presented in this chapter.

5.1 Overview

The synthesis described in the previous chapter allows us to synthesize controllers for MSD specifications where MSDs have concrete lifelines that refer to objects in a static object system. The synthesis is therefore not suited for finding inconsistencies in large, dynamic systems, which we need to specify with MSDs that have symbolic lifelines. However, *single use cases* in such systems typically refer to a *static structural context*, which means that they describe how objects that have certain relationships to each other shall interact. In the following, we also say that use cases *occur* among certain objects in an object system. It is therefore possible to specify the behavior for a single use case based on a particular small and static object system that represents subsets of objects among which the use case may occur in a larger object system. A single use case can, thus, be specified in the form described in Sect. 4.2, and can then serve as input for the synthesis.

If such a specification, which we call a *use case specification* in the following, turns out to be inconsistent, this means that any specification containing the use

case specification will be inconsistent. The engineers will then have to resolve this inconsistency. However, if all the use case specifications are consistent, this does not necessarily mean that a specification consisting of all these use case specifications is also consistent. It may still happen that *occurrences* of the use case *overlap*, which means that active MSDs of different use cases may occur in such a way that their lifelines bind to the same objects and express requirements about how certain messages shall be exchanged among these objects. These requirements may still be contradictory in some cases: an active MSD in one use case occurrence may require something to happen that is forbidden by an active MSD in another use case occurrence.

An illustration of use cases that occur among certain objects is shown in Fig. 2.1. Here, for example, the occurrences of the use cases *Drive onto merging switch* and *Enter denied when hazard on next track section overlap*.

Typically, it is difficult to anticipate in which combinations use cases may overlap. One way to find that out is by using the simulation of the complete specification via the play-out algorithm. When simulating a larger, possibly dynamic system, the play-out algorithm can interpret the MSDs in the use case specifications as symbolic MSDs (provided that adequate binding expressions are specified for the lifelines in the MSDs).

The play-out algorithm could, however, run into avoidable violations during the play-out. This means that the engineer will not know whether a hot violation during the simulation indicates an inconsistency or not. However, the controllers synthesized from the above use case specifications contain information of how to execute the MSDs belonging to the same use case so that violations can be avoided. The idea is therefore to exploit the information in these controllers to reduce the number of avoidable violations during the play-out. If, during play-out, we can track which active MSDs make up a certain *use case occurrence*, then we can control the play-out of these active MSDs according to the controller that was synthesized from this use case specification. This way it can be ensured that the play-out algorithm takes no steps that would lead to a violation in the active MSDs in that use case occurrence.

How to formally specify use case specifications and how to control the execution of active MSDs of a use case occurrence according to a controller that was synthesized from the use case specification is explained in Sect. 5.2.

Despite controlling the simulation by the controllers that were synthesized from the use case specifications, violations may occur during the simulation where use case occurrences overlap. Again, these violations may indicate an inconsistency or an avoidable violation. To find this out, we would like to analyze such overlappings of use case occurrences in more detail. Also overlappings of use case occurrences often take place in a static structural context. In that case, we can again employ the synthesis to determine whether violations of the requirements can always be avoided or not. For this purpose, the overlapping of use case occurrences in a certain structural context must first be captured formally.

Section 5.3 describes how a situation where use cases overlapping in a certain structural context can be specified in what is called a *composed use case specification*. The section describes furthermore how controllers can be synthesized

from these specifications. If these specifications turn out to be consistent, the section explains how *occurrences* of such composed use cases can be tracked and how the execution of the involved active MSDs can be guided by the synthesized controllers in order to avoid yet more avoidable violations during further simulations.

Figure 5.1 summarizes the design process that shall be supported by the technique presented in this chapter. After the use cases are described informally (1), the use case descriptions must be captured formally in use case specifications (2). Sometimes already typical overlappings of use cases are known to the engineer at design time. If that is the case, this knowledge is very valuable and should be capture in the form of composed use case specifications right away. Next, controllers can be synthesized from the (composed) use case specifications (3).

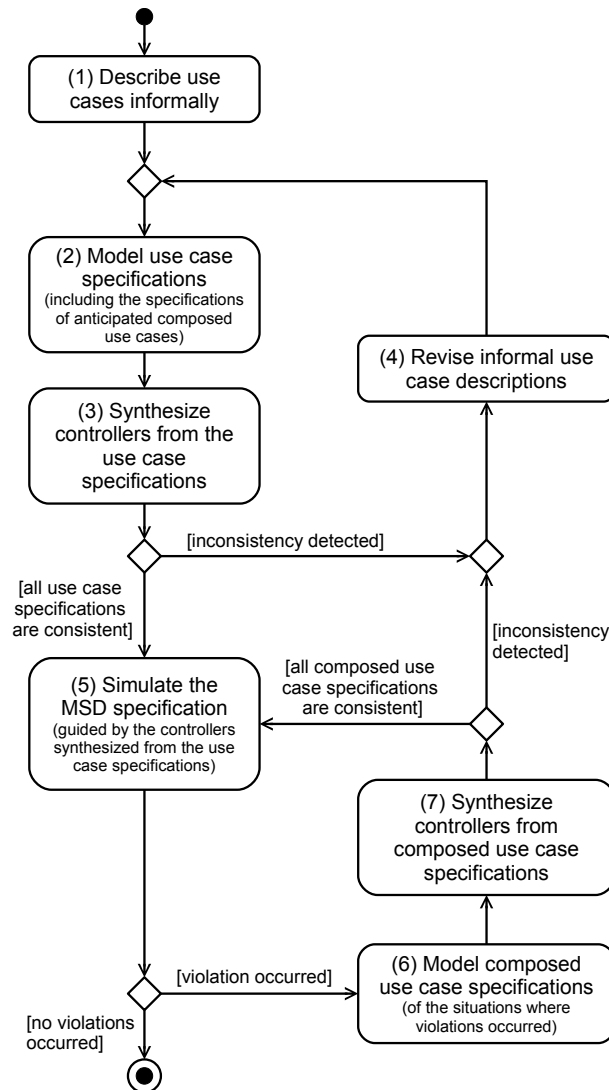


Figure 5.1: The process of finding and resolving inconsistencies in scenario-based specifications envisioned in SCENARIOTOOLS

If inconsistencies are detected during the synthesis, this means that the specification is inconsistent, and the engineers will have to revise the use case descriptions (4). If controllers can be successfully synthesized from the use case specifications specified so far, but it may be that use cases occur in combinations that were not anticipated before, the simulation via the play-out algorithm can be used, guided by the synthesized controllers in order to minimize the amount of avoidable violations (5). If violations during the simulation occur, the overlapping of use case occurrences in which this violation occurs can be formally captured in a composed use case specification (6), and synthesis can determine whether violations are always avoidable in this situation (7).

If it turns out that the composed use case specification is inconsistent, the requirements must again be revised (4). Otherwise, if the synthesis is able to synthesize a controller from the composed use case specification, this controller can again be used to improve the simulation, such that the avoidable violation can be avoided during the next simulation run (5). If no violations occur during the simulation, there is an increased probability that the specification is consistent. That probability, however, depends on how intensively the simulation was carried out, i.e., how many of the possible overlappings of use cases were simulated and how many of the possible sequences of environment events in such situations were simulated.

This approach is of course limited if many use cases may overlap in various ways. Then the synthesis of the composed use cases may not be feasible anymore. Also, as we will see in Sect. 5.3, the process of tracking composed use case occurrences for guiding the play-out of the involved active MSDs may become very complex and may have a negative effect on the performance of the simulation.

5.2 Guiding play-out by controllers synthesized from single use cases

This section first introduces an example use case specification in Sect. 5.2.1 and then explains in Sect. 5.2.2 how the play-out algorithm can be improved by controllers that could be successfully synthesized from use case specifications.

5.2.1 Use case specification example

Let us first consider an example use case specification in the following. Figure 5.2 shows the specification of a use case which describes that when a RailCab detects an obstacle on the track section, it must report to its current track section control that a hazard has occurred at a certain position on the track section. The track section control then has to inform other RailCabs driving on the same track section about the occurrence of a hazard on the respective position.

Use case specification overview

In the collaboration diagram at the top, Fig. 5.2 shows the four objects that are involved in the use case: the environment, a RailCab that detects an obstacle,

a track section control, and a RailCab that is warned about the obstacle. The roles in the collaboration diagram are typed over the classes shown in the class diagram of Fig. 3.10.

The MSDs are a deliberately odd formalization of the use case in order to construct a small example where naive play-out may run into an avoidable violation. The requirements are described by three MSDs. The MSD `WarningWhenObstacleDetected` says that if the detecting RailCab detects an obstacle, it must notify the track section control of a hazard that occurred on the track section. The track section control must then inform the warned RailCab about the hazard and, next, must tell it about the position of the hazard. (The actual position would be represented by a parameter, but, as mentioned previously, the synthesis does not yet support message parameters. So, this example abstracts from the concrete position.)

The MSD `ReportObstaclePosition` describes a similar scenario. When the detecting RailCab detects an obstacle, it must inform the track section control about the position of the obstacle. The track section control must then send this position to the warned RailCab. Last, the MSD `ReportObstaclePositionAndIssueWarning` says that if the detecting RailCab notifies the track section control about a hazard and then sends the track section control the position of an obstacle, the track section control must inform the warned RailCab about the position of an obstacle and then warn it about a hazard.

Violating and non-violating executions

During the play-out of these MSDs, the play-out algorithm may run into a violation. When the event `obstacleDetected` occurs, and the RailCab then sends `hazardOccurred` and then `obstaclePosition`, the message `hazardWarning` is active in the MSD `WarningWhenObstacleDetected` and the event `obstacleAtPosition` is active in the other two MSDs. But, at this point the play-out algorithm will not be able to progress, because `obstacleAtPosition` is forbidden to occur in `WarningWhenObstacleDetected` and `hazardWarning` is forbidden to occur in `ReportObstaclePositionAndIssueWarning`. If the play-out algorithm, however, would have chosen to send `obstaclePosition` before `hazardOccurred`, it could have avoided this violation.

The use case specification can serve as input for the synthesis described in Chap. 4, and an admissible controller can be synthesized from this specification, in which the violating execution is avoided.

Play-out of the MSDs

For the synthesis, the collaboration diagram describes one simple, static object system. But, it is possible to model various, larger track systems based on the classes that the roles refer to, as shown at the bottom of Fig. 3.10. Based on such a larger object system, the play-out algorithm can also interpret these MSDs as symbolic MSDs. (Possibly in combination with other MSDs from further use case specifications.) For this purpose, however, as described in Sect. 3.1.15,

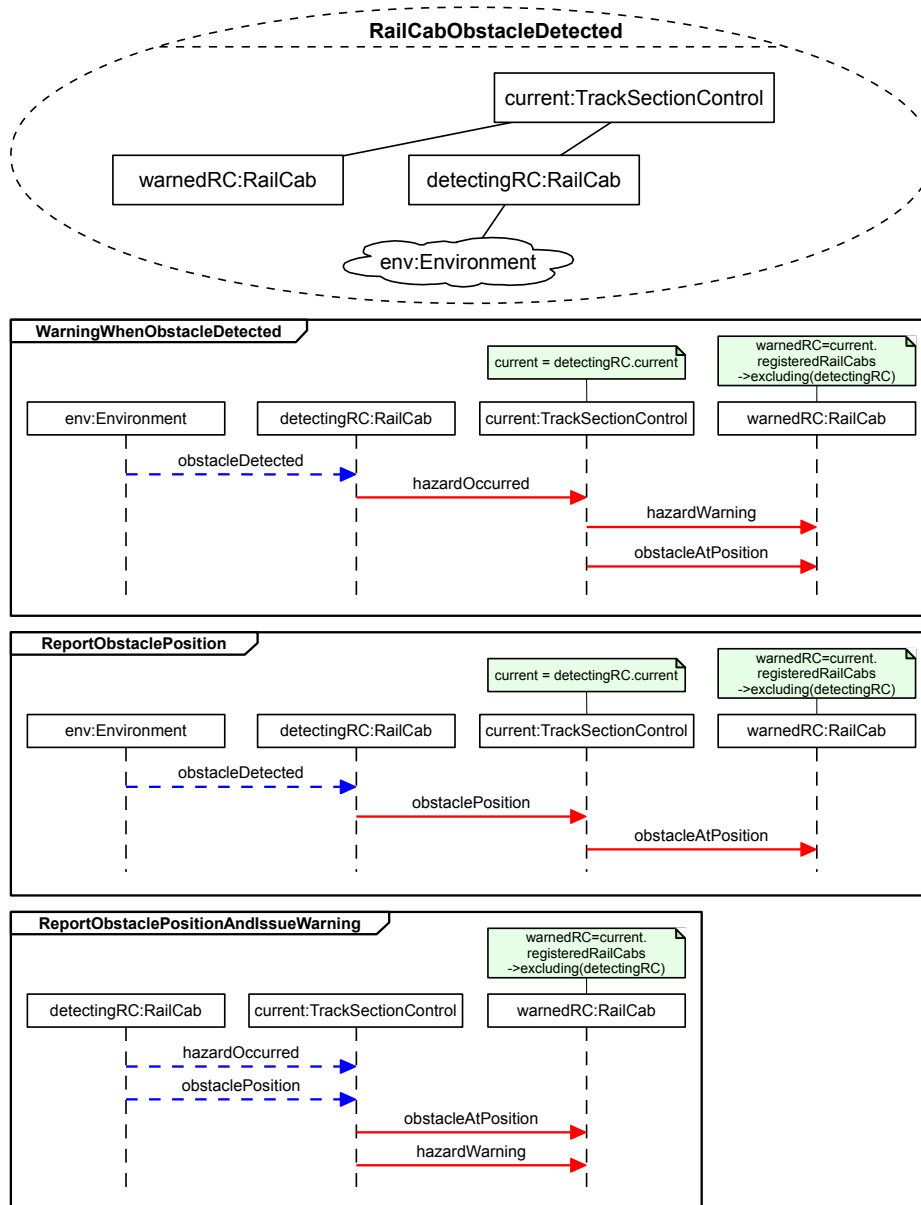


Figure 5.2: The collaboration diagram and MSDs of the Warn RailCabs on track use case specification

binding expressions are required for the lifelines that will not be bound during the unification of the first diagram message, in order to specify which objects these lifelines shall be bound to.

The MSDs in Fig. 5.2 have binding expressions for these lifelines. In the MSD `WarningWhenObstacleDetected`, for example, the binding expressions say that the track section control bound to the lifeline `current:TrackSectionControl` must be the current track section control of the detecting `RailCab`. Furthermore, the lifeline `warnedRC:RailCab` shall be bound to any `RailCab` that is registered at the current track section control and that is not the detecting `RailCab`. Remember that if the value expression of a binding expression evaluates to a set of two or more objects, copies of the active MSD will be created so that every object is bound to the lifeline in one active MSD, see Sect. 3.1.15.

The binding expressions are the same for the according lifelines in the other two MSDs, which is not surprising, since the lifelines represent the same objects in the use case. Because the binding expressions are very often redundant, it would be desirable in the future to specify these binding expressions only once, for example in the collaboration diagram. The `SCENARIOTOOLS` implementation of the play-out algorithm, however, currently requires that the binding expressions are specified within the MSDs.

5.2.2 Play-out with synthesized controllers

This section presents the extended play-out algorithm. Before coming to the extension, the concept of a *use case occurrence* is introduced. Furthermore, it is explained how the information of which steps are allowed to be performed by the system is extracted from the synthesized controllers.

The use case occurrence

During runtime, a *use case occurrence* is the set of all active MSDs where, first, the MSDs belong to the same use case, and, second, all lifelines representing the same role are bound to the same object. In a use case occurrence, the lifeline bindings also imply which object is represented by a role. The mapping of a role to an object is also called a *role binding*.

The synthesized controller

The controller created by the synthesis described in Chap. 4 is an automaton that contains directives of which transitions to take in each winning state of the TGA network that corresponds to this use case specification. A state in this controller is determined by the current locations of all processes in the TGA network as well as the current value of all variables. This includes the lifeline variables, which encode the current cuts of the MSDs.

In the following, however, we are not interested in every state of the synthesized automaton. We only focus on certain states and transitions in this controller: if we talk about a *state* in the following, we refer to a state where the system automaton is in the location `systemActive`. The outgoing transitions in these states represent the directive to either execute a certain event in

a particular configuration of cuts, or to wait for environment events to occur. The former transitions are called *event transitions*, the latter are called *waiting transitions*. The event transitions are transitions where the system automaton takes an edge to the location `produceEvent` and assigns a particular value to the `event` variable. (See Sect. C.2.4 for an example of the output generated by UPPAAL TIGA for the synthesis of the use case specification above.) If a state has no event transitions, there is a waiting transition instead, where the system automaton moves to the location `systemInactive`.

The value assigned to the `event` variable in an event transition represents a message that refers to a certain operation and is sent between the objects that are represented by the roles in the use case specification (see Sect. 4.3.1). We say that a message in the object system can be *unified* with an event transition if the operation specified by the event transition equals the operation of the message in the object system and if the sending and receiving roles specified by the event transition are bound to the sending resp. receiving object of the message sent in the object system.

The extended play-out algorithm

The idea is now that one controller or even several controllers that were synthesized from different use case specifications tell the play-out algorithm by which steps it will be able to avoid violations among the MSDs that belong to an according use case occurrence. This works by “obeying” the controller for specific use case occurrences.

Remember that in an active MSD, in a certain cut, there may be one or more messages active. These messages determine which messages shall be sent between the objects bound to the sending and receiving lifelines. If a message occurs in the object system that can be unified with a message that is not enabled in the current cut, this is a cold violation or a safety violation, depending on the temperature of the cut. Events that lead to safety violations are called forbidden events. In a step, the play-out algorithm chooses to send a message in the object system that is unifiable with a message that is active in some active MSD, provided that it is not forbidden in another active MSD (see Sect. 3.1.8).

If there is an occurrence of a use case for which a controller exists, also called a *controlled* use case occurrence, a similar principle is additionally applied for the controller: In a certain state of a use case occurrence, i.e., in a certain configuration of cuts of the active MSDs that make up the use case occurrence, there may be a number of outgoing event transitions. The events represented by an outgoing event transition in the current state are also called *prescribed events*. If a message occurs in the object system that is not prescribed by a controller, but can be unified with another event transition in the controller, this is called *disobeying* the controller; events which disobey the controller in a certain state are also called *disobeying events*.

When executing active events, the play-out algorithm will always try to execute events that are not disobeying any controlled use case occurrence. Disobeying a controlled use case occurrence is not necessarily a violation of the requirements, but it means that from that point on the controller cannot guar-

antee that it will be always possible to avoid violations of the MSDs within the use case specification.

More specifically, the extended play-out algorithm works as follows.

0. Wait for an environment event to occur. If an event occurs that is forbidden by a currently active MSD, this is a hot violation. Then terminate. If an environment event occurs that is not forbidden in any active MSD, then continue as follows.
1. For all controlled use case occurrences, map the current cuts of the active MSDs in that use case occurrence to a state in the controller. Mark this state as the *current* state of the controller automaton for that use case occurrence.
2. For each controlled use case occurrence, determine the set of prescribed and disobeying events in the current state of the controller automaton for the use case occurrence.
3. Determine the set of events that are active in an active MSD. Then,
 - a) if this set is empty, do nothing and wait for the next environment event to occur (return to Step 0). Otherwise
 - b) remove from this set of events those that are forbidden to occur in an active MSD. If the resulting set is empty, this constitutes a hot violation, i.e., there are steps that have to be executed, but there is no step that can be taken that would not lead to a safety violation of an active MSD. The algorithm is terminated and an according exception is thrown.
 - c) Remove from the remaining set of events the events that are disobeying a controller of a controlled use case occurrence.
 - If the resulting set is not empty, choose one of the events and execute it. Then continue with Step 1.
 - If the resulting set is empty, execute an event from the previous set (the set of active, non-forbidden events) and issue a warning that the event is disobeying a controller and that from now on the play-out will not be able to avoid violations among the active MSDs within every controlled use case occurrence. Then continue with Step 1.

(It is assumed that when events are executed or environment events occur in the environment, the cuts of the MSDs progress as in the regular play-out algorithm.) Figure 5.3 illustrates this process. On the left of the figure, a use case occurrence is illustrated with a set of active MSDs that all reside in a certain cut. The dashed lines from the use case ellipse to the environment, the RailCabs, and the track section control illustrated at the bottom express which role, and consequently, which lifeline is bound to which object. On the right, the steps of the play-out algorithm as explained above are illustrated.

Currently, message parameters and diagram variables are not supported by the synthesis explained in Chap. 4, and the SCENARIOTOOLS play-out algorithm does not yet support the play-out of timed MSD specifications and assumption MSDs. So, the symbiosis of synthesis and simulation is currently only supported for the core MSD language features. When extending these concepts to parameterized messages and time, it will also be necessary to consider the current value

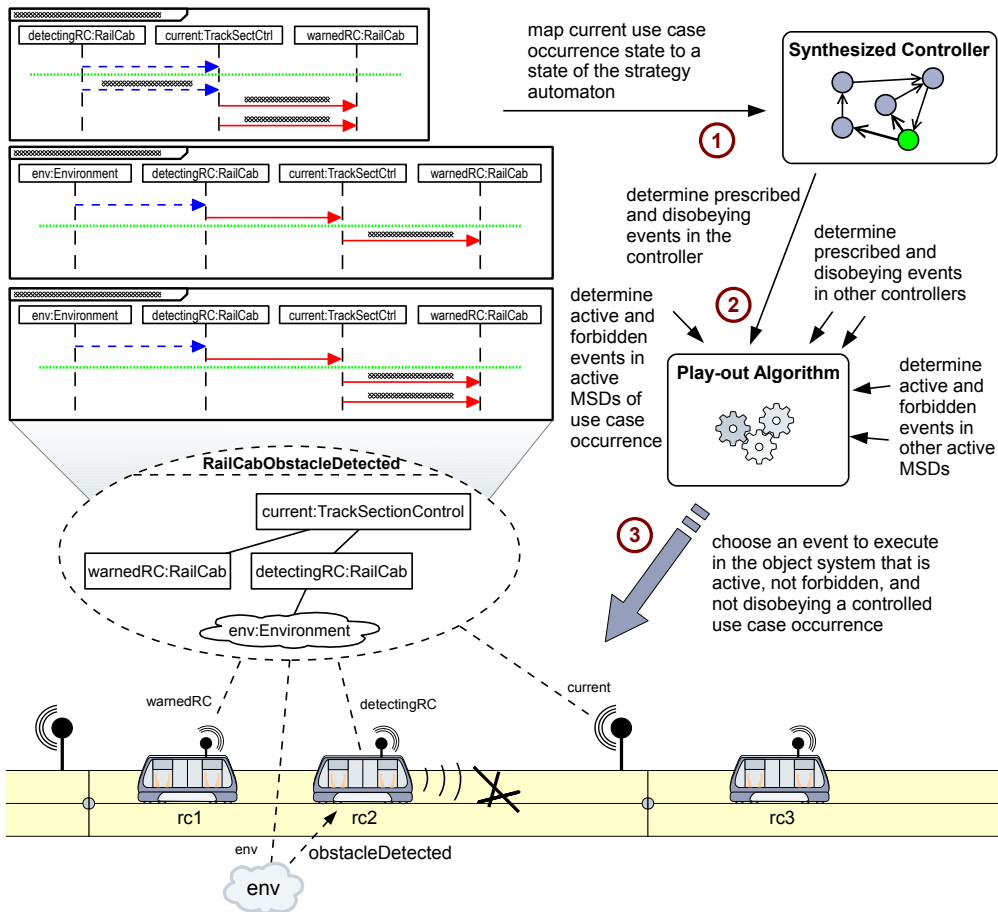


Figure 5.3: Illustration of how the play-out of the active MSDs belonging to a use case occurrence is aided by a synthesized controller strategy

of diagram variables and clocks when mapping the current state of the active MSDs in the use case occurrence to the current state of the controller (Step 1).

Complete controllers

For this process, *complete* controllers must be synthesized from the use case specifications. A complete controller is a controller that contains all winning actions in each winning state of the TGA network. In the case of a controller synthesized from a use case specification, a complete controller contains all admissible reactions of the system objects in a certain configuration of cuts in the use case occurrence, i.e., all possible steps that guarantee for the execution of the MSDs within the use case specification to always avoid hot violations and to always eventually execute all the active events. If the controllers are not complete, the play-out algorithm would consider too many events as disobeying events, and would therefore report warnings where there would still be a chance to execute the MSDs in an admissible way. This would contradict the desired goal of reducing the avoidable violations, or “false negatives”, that are reported

to the engineer. Fortunately, UPPAAL TIGA allows us to synthesized complete controllers [BCD⁺07b].

Iterative vs. invariant interpretation of the MSDs

Another issue is that the synthesis only supports the *iterative* interpretation of the MSDs, which means that the controller only considers states where there is at most one active copy of an MSD in the use case specification (see Sect. 3.1.6). The play-out algorithm must therefore also follow the iterative interpretation, or, if it supports the invariant interpretation of the MSDs, it must be ensured that during play-out never a situation occurs where two active copies of the same MSD are created within the same use case occurrence. Only in these cases, the iterative and invariant interpretations lead to the same behavior. The play-out algorithm implemented in SCENARIOTOOLS supports the invariant interpretation, but no examples of MSDs occur in the scope of this thesis where the invariant interpretation differs from the iterative interpretation.

There are never two active copies of an MSD created if the first message never appears again in the diagram. Cases where the invariant interpretation could differ from the iterative interpretation can therefore be identified by a simple syntax check.

Late binding of lifelines

The problem with guiding the execution of the active MSDs belonging to a use case occurrence as described above is that not all lifelines of an active MSD may be bound right at the time of its activation. The binding may take place gradually as the active MSD progresses. A binding expression may for example refer to a diagram variable that is bound to the value of a parameter of a parameterized message that appears later in an MSD. If a lifeline is not bound right after the activation of an MSD, this is called the *late binding* of lifelines.

If there is late binding, it is not always clear which use case occurrence an active MSDs will turn out to belong to. However, if that is not clear, it is not possible to determine which state a use case occurrence is in, and the play-out cannot be guided by a controller as explained above. Therefore, late binding is forbidden in this approach.

Because parameterized messages, diagram variables, and object properties are not yet supported by the synthesis, no late binding can currently occur in the use case specifications.

An active MSD could belong to multiple use case occurrences

One case that may occur, even without late binding, is that an active MSD could belong to multiple use case occurrences. Imagine that there are two use case occurrences of the same use case where a subset of the same roles are bound to the same objects. An MSD with lifelines that only refer to these roles would, after its activation, belong to both use case occurrences.

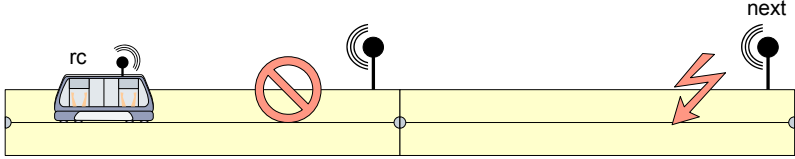
5.3 Guiding play-out by controllers synthesized from composed use cases

This section first explains how to model composed use case specifications by an example in Sect. 5.3.1. Then Sect. 5.3.2 explains how a composed use case specification can be mapped to the input for the synthesis as described in Chap. 4. Next, Sect. 5.3.3 illustrates how the active MSDs that belong to the occurrence of a composed use case can be controlled by a controller that was synthesized from the composed use case specification. Last, Sect. 5.3.4, Sect. 5.3.5, and Sect. 5.3.6 explain how to systematically track the active MSDs that make up composed use case occurrences during the simulation.

5.3.1 Composed use case example

Consider the use case Enter denied when hazard on next track section, which is informally described in Tab. 5.1. It says that a RailCab that requests the permission to enter the next track section must not be allowed to enter if a hazard has occurred on that next track section.

Table 5.1: Use Case Enter denied when hazard on next track section

Use Case: Enter denied when hazard on next track section	Nr. 4
Requirements: If a RailCab requests the permission to enter the next track section from the next track section control, but there is a hazard on the next track section, the next track section control must deny the permission to enter.	
Environment assumptions: none	
Sketch: 	

The specification of the use case Enter denied when hazard on next track section is shown in Fig. 5.4. It says that if a RailCab requests the permission to enter the next track section, and the hazard flag is set to true, the next track section control must not allow the RailCab to enter.

When formulating this use case, the engineer could have in mind that a hazard occurred on the next track section control because another RailCab on that track section detected an obstacle, similar to the example in Sect. 5.2.1. Furthermore, that a RailCab requests the permission to enter the next track

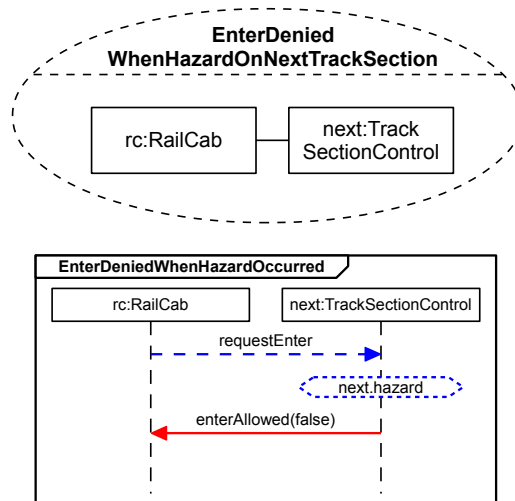


Figure 5.4: The collaboration diagram and the (single) MSD specified for the use case Enter denied when hazard on next track section

section is a behavior that was specified in the use case Drive onto track section. Figure 5.5 shows a picture of the situation that the engineer could have in mind.

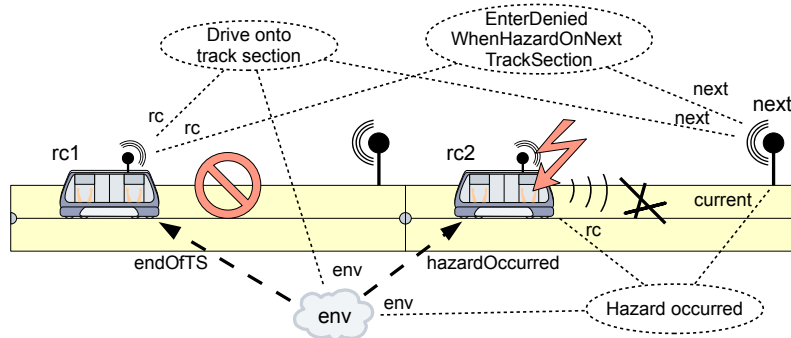
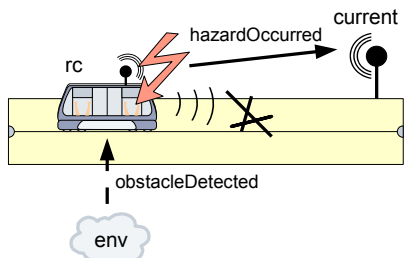


Figure 5.5: A combination of the use cases Drive onto track section and Hazard occurred and Enter denied when hazard on next track section in a particular instance situation

The use case Hazard occurred is another simple use case that is described in Tab. 5.2. The use case specification is shown in Fig. 5.6. Here it is specified that if a RailCab detects an obstacle on the track section, it reports to the current track section control that a hazard has occurred on the track section. The track section control must then set its hazard property to true, which is modeled by the self-message `setHazard(true)`. (Remember that set-messages have side-effects on object properties, see Sect. 3.1.11.)

In the situation illustrated in Fig. 5.5, a combination of use cases overlap in a particular structural context. This situation can be modeled by a UML collaboration diagram as shown in Fig. 5.7. In the following, we call the specification of such a situation a *composed use case*. This composed use case is called

Table 5.2: Use Case Hazard occurred

Use Case: Hazard occurred	Nr. 3
<p>Requirements: If a RailCab detects an obstacle on the track section, it must report that a hazard occurred to its current track section control.</p>	
<p>Environment assumptions: none</p>	
<p>Sketch:</p> 	

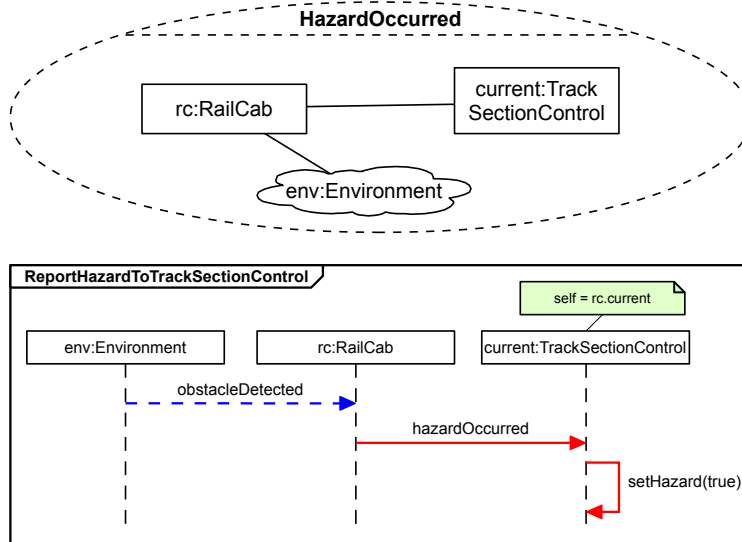


Figure 5.6: The collaboration diagram and the (single) MSD for the use case Hazard occurred

Enter denied when hazard on next track section AND Drive onto track section AND Hazard occurred. Here the objects among which the use cases occur are again modeled by roles. We see the environment, two RailCabs, and one track section control. (To reduce the visual complexity, no connectors are shown among the roles in this diagram.) In addition, the collaboration diagram shows the occurrences of use cases among these roles by the dashed ellipses. In UML, this notation is called a *collaboration use* [UML09, Sect. 9.3.4, pp. 171]. Here we call these ellipses *internal use case occurrences* instead.

Internal use case occurrences refer to other use case specifications as indicated by their name. The dashed lines between the ellipses of internal use case occurrences and the roles describe which role in the use case specification that the internal use case occurrences refer to is mapped to which role in the composed use case. These mappings are also called *role mappings*. There can only be role mappings if the class that types the role in the specification of the internal use case is equal to or a generalization of the class that types the role in the composed use case.

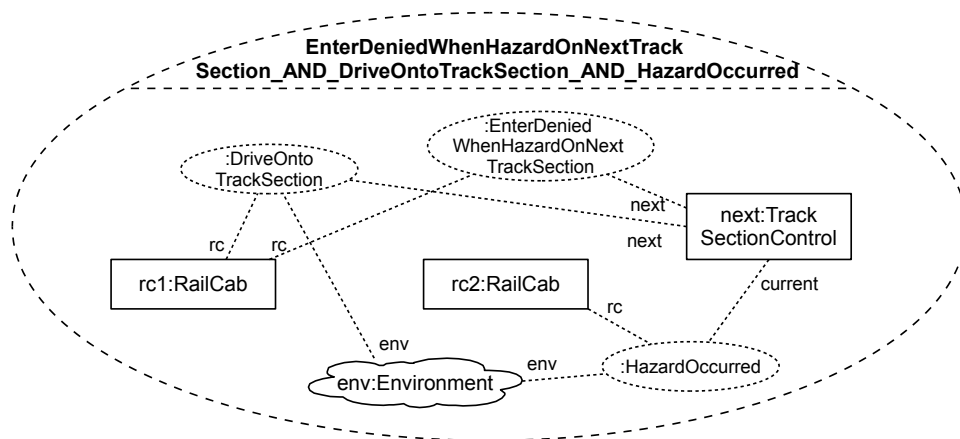


Figure 5.7: The collaboration diagram for a use case that is composed of occurrences of the use cases Drive onto track section, Hazard occurred, and Enter denied when hazard on next track section

5.3.2 Synthesizing controllers from composed use cases

A composed use case specification as illustrated above can be mapped to a regular use case specification that can then serve as the input for the synthesis described in Chap. 4. In this mapping, each lifeline of each MSD in each internal use case is modified so that it does not anymore reference the role in the internal use case specification, but instead references the role in the composed use case that the role in the internal use case is mapped to. It may, however, happen that there are two or more internal use case occurrences of the same use case within a composed use case with different role mappings. Then it is necessary that the MSDs in the internal use case are copied for each internal use case occurrence. The lifelines in each MSD copy for an internal use case occurrence must then

refer to the role in the composed use case according to the role mapping of the internal use case occurrence.

Figure 5.8 illustrates this mapping by the example. The left side shows a composed use case 2OT, which specifies that the use case Drive onto track section (here abbreviated OT) occurs twice within a certain structural context. On the top left, there are two MSDs with lifelines that refer to the roles in the specification of the use case OT, indicated by the lifeline names. On the right, the figure displays the corresponding “flattened” use case specification. The collaboration diagram contains the same roles as the collaboration diagram of the composed use case, but there are no internal use case occurrences left. Instead, there are two copies of the MSDs in the use case specification OT where the lifelines refer to the roles according to the role mappings of the internal use case occurrences on the left.

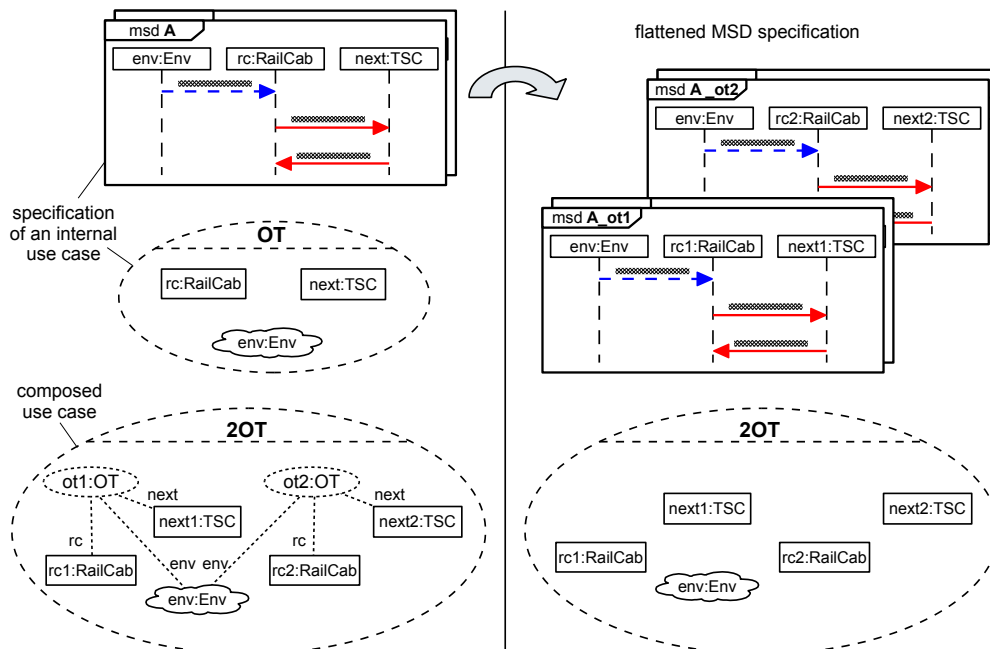


Figure 5.8: “Flattening” composed use cases to the input for the synthesis

Composed use cases may also again be composed to composed use cases. The mapping to a “flattened” use case occurrence then works by iteratively applying the mapping described above.

5.3.3 Guiding the play-out of composed use case occurrences

If a controller could be successfully synthesized from a composed use case, it is possible to control the MSDs that make up the occurrence of that composed use case according to the synthesized controller. To do that, it is necessary to determine which active MSDs make up occurrences of use cases that are internal use cases of the composed use case. Then it has to be determined

which combination of these use case occurrences makes up an occurrence of the composed use case.

Figure 5.9 illustrates a state during the simulation of the RailCab system where the composed use case `Enter denied when hazard on next track section AND Drive onto track section AND Hazard occurred` (see Fig. 5.7) occurs. The state that is illustrated here is the state after RailCab `rc2` has detected an obstacle on the track section and, as specified in the MSD `ReportHazardToTrackSectionControl` in Fig. 5.6, reported a hazard to its current track section control, which then set its `hazard` flag to true. Then, RailCab `rc1` was notified about reaching the end of its current track section and, in reaction, sent the message `requestEnter` to its next track section control. The top of the figure shows three active MSDs that make up occurrences of the use cases `Drive onto track section` and `Enter denied when hazard on next track section`. Currently, there is no active MSD of the use case `Hazard occurred`, because it was already terminated after the `hazard` flag was set to true. The lifelines of the active MSDs are bound as indicated by the labels in the rounded rectangles that are attached to the lifelines.

The lifeline bindings imply which roles in the use case occurrences are bound to which objects. According to the role mapping in the composed use case, these use case occurrences make up an occurrence of the composed use case where the role `rc1` is bound to the object `rc1`, the role `rc2` is bound to the object `rc2`, the role `env` is bound to the object `env`, and the role `next` is bound to the object `tsc4`. These role bindings are indicated by the dashed lines to the objects illustrated at the bottom of the Fig. 5.9.

The current configuration of cuts of the active MSDs that belong to the composed use case occurrence can be mapped to a state in the controller automaton. The play-out can then be guided by the controller as described in Sect. 5.2.2.

5.3.4 Tracking composed use case occurrences

One difficulty in this approach is, however, that the life-cycle of a composed use case already starts even if just one use case occurs that is an internal use case of the composed use case. In the above example, already the occurrence of the use case `Hazard occurred` was actually an occurrence of the composed use case. Therefore, in order to control the play-out of the involved MSDs so that play-out takes no steps that may lead to an avoidable violation in a composed use case later on, already these occurrences of single use cases that occur internally in a composed use case have to be controlled according to the controller that was synthesized from the composed use case. That means that we have to control the play-out of these use case occurrences long before occurrences of all use cases occurring internally in a composed use case finally aggregate to make up the situation that is specified in the composed use case. Even worse, there are often various ways in which the situation specified in the composed use case may emerge from this occurrence of one internal use case. Let us consider an example in the following.

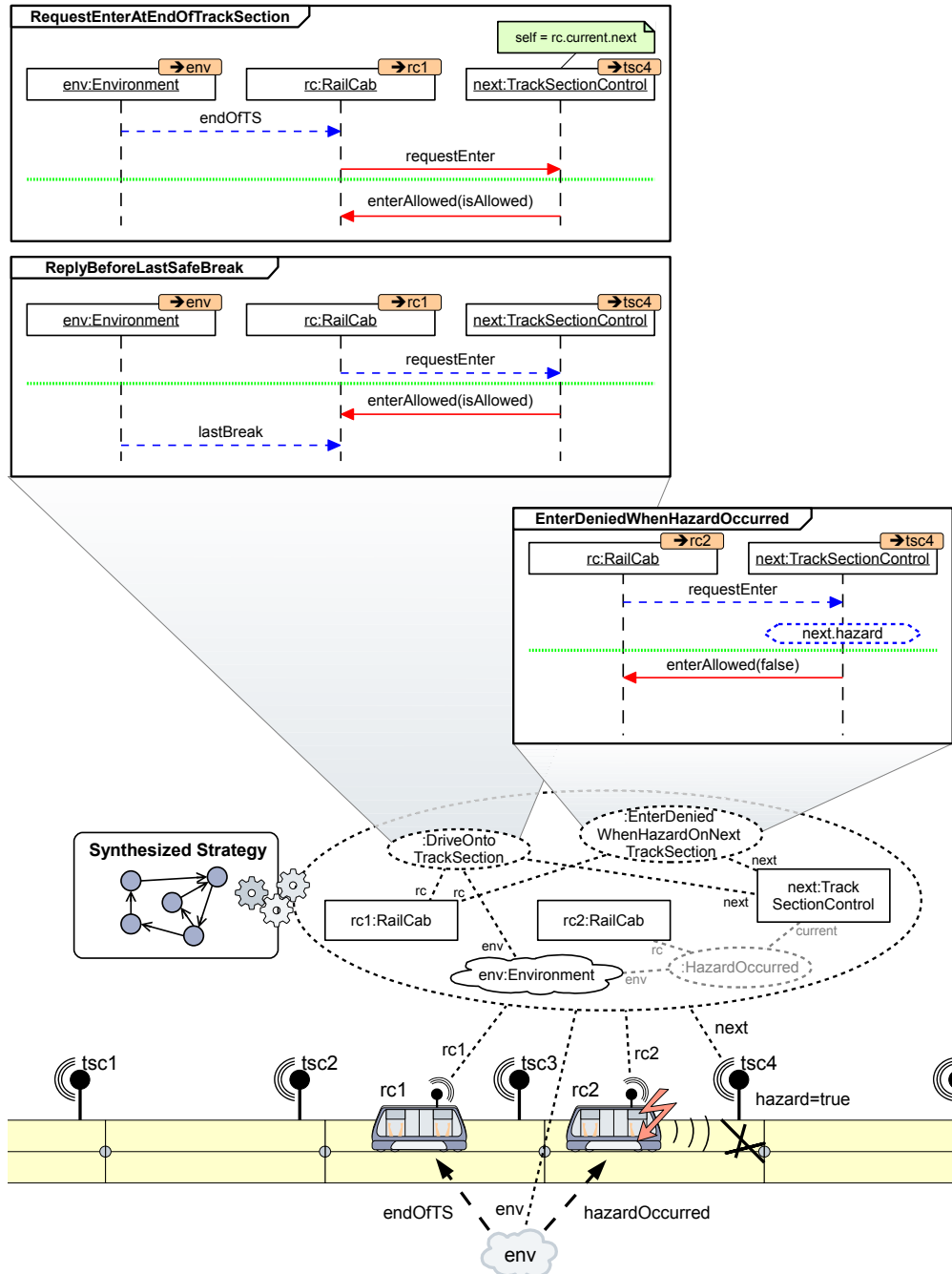


Figure 5.9: An occurrence of the composed use case Enter denied when hazard on next track section AND Drive onto track section AND Hazard occurred that is executed based on a synthesized controller strategy

Example of tracking composed use case occurrences

The top of Fig. 5.10 shows the collaboration diagram of a composed use case Drive onto track section AND Drive onto track section, in the following abbreviated 2OT. This composed use case describes a situation where two RailCabs on two track sections reach the end of the track section and prepare for entering the next track section. The diagram shows that the use case contains two internal occurrences of the use case Drive onto track section. One is named *ot1*, with its roles *env*, *rc*, *next* mapped to the roles *env*, *rc1*, *next1*, respectively, of the composed use case. The other is named *ot2*, with its roles *env*, *rc*, *next* mapped to the roles *env*, *rc2*, *next2* of the composed use case respectively¹.

The bottom of the figure shows two RailCabs, *rcA* and *rcB*, that drive on different track sections in a RailCab system. Imagine that now *rcA* approaches the end of its track section and there is an occurrence of the use case Drive onto track section. This use case occurrence is called *a* and is represented in Fig. 5.10 by the dashed ellipse that is drawn just above the RailCab *rcA*. The occurrence of this use case is already an occurrence of the composed use case. Actually, this use case occurrence resembles two occurrences of the composed use case, because the use case occurrence *a* can take the role of the internal use case occurrence *ot1* or the internal use case occurrence *ot2* of the composed use case. These two occurrences of the composed use case are represented in Fig. 5.10 by the two dashed ellipses on the left in the row labeled L1.

If a use case occurrence plays the role of an internal use case occurrence in a composed use case occurrence, this use case occurrence is also said to be *bound* to the internal use case occurrence. If there is no use case occurrence that takes the role of an internal use case occurrence within a composed use case occurrence, the internal use case occurrence is *unbound*. The label $[(ot1, \underline{a}), (ot2, ?)]$ for example says that the use case occurrence *a* is bound to the internal use case occurrence *ot1* and that the internal use case occurrence *ot2* is unbound.

If in a composed use case occurrence all internal use case occurrences are bound, it is called a *complete* composed use case occurrence, otherwise it is an *incomplete* or *partially bound* composed use case occurrence.

Next, let us assume that the RailCab *rcB* also reaches the end of its track section. That means that the use case Drive onto track section occurs another time, which is illustrated in Fig. 5.10 by the use case occurrence named *b*. Here two incomplete occurrences of the composed use case are created as in the first case. In addition, a complete occurrence of the composed use case can now be formed by binding *ot1* to *a* and *ot2* to *b*. But, alternatively also *ot2* can be bound to *a* and *ot1* to *b* (see the two ellipses in the row L2).

In order to keep the play-out algorithm at all times from taking steps that may lead to an avoidable violation in a composed use case occurrence, the play-out must not take any steps that are forbidden by the controller synthesized from

¹Note that the two internal use case in this composed use case actually do not overlap, i.e., they do not specify requirements for messages sent among the same roles. Therefore, if the use case Drive onto track section is consistent, the synthesis of the composed use case would not find any inconsistencies. But this is just a simple example with the purpose to illustrate how to track occurrences of the composed use cases during the simulation.

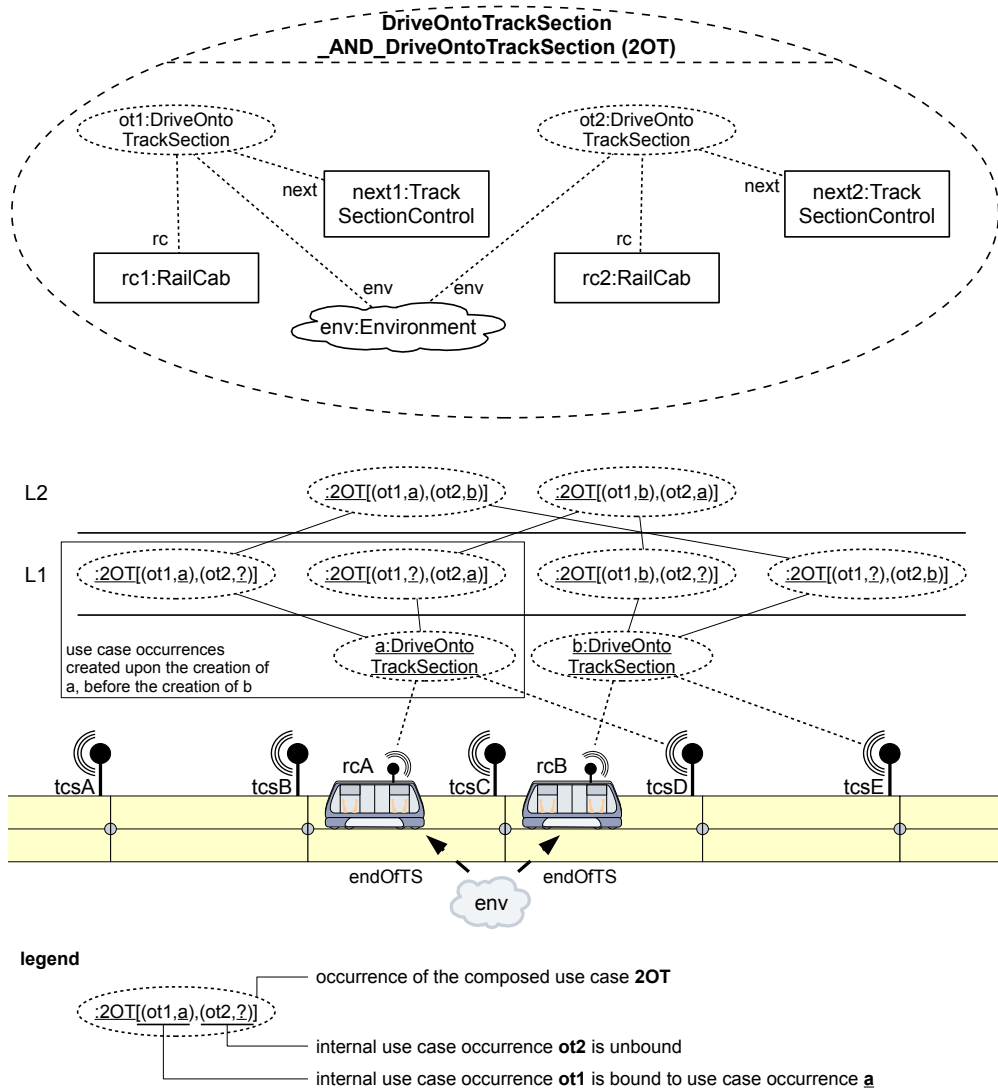


Figure 5.10: An example of incomplete and complete occurrences of a composed use case

the composed use case in the state that corresponds to the current configuration of cuts in the active MSDs that make up any of the above composed use case occurrences. We also say that the play-out must *obey* the controller in all the above composed use case occurrences. It must even obey the partial composed use case occurrences, because the use case *Drive onto track section* may occur in the system another time to form a further complete use case occurrence with the use case occurrences a and b.

Structural constraints among roles of composed use cases

In this example, there is an explosion of the number of composed use case occurrences that need to be tracked the more often the use case *Drive onto track section* occurs. One reason for that is that the composed use case does not describe whether the internal use case occurrences shall occur in any particular *structural context*. Imagine that we actually like to capture a situation where the two use cases *Drive onto track section* occur on two *subsequent* track sections. If we could specify such structural contexts, and then only track composed use case occurrences within these structural contexts, we could reduce the combinations of use case occurrences that we would need to consider during the simulation.

In the following, a concept is presented that allows the engineer to specify structural relationships between roles via OCL expressions, similar to the binding expressions on lifelines. We call these OCL expressions *context expressions* in the following. If such expressions are specified, we only track use case occurrences for which these expressions evaluate to true.

Figure 5.11 shows the expression `next1.next=next2` inside a label in the collaboration diagram of the composed use case 2OT. These expressions must evaluate to a truth value. In these expressions, the role names can be used as variables. If the roles are bound, these variables are bound to the according objects; if the roles are unbound, the variables are unbound. Expressions can only be evaluated if all variables appearing in the expression are bound. The above expression says that the object bound to the role `next1` must link to the object bound to role `next2` as its next track section control.

In Fig. 5.11, the row labeled L2 shows that now the composed use case occurrence with the internal use case bindings `[(ot1,b),(ot2,a)]` is not considered anymore.

The next section describes more precisely how use case occurrences are tracked systematically and how the context expressions are interpreted in this process.

5.3.5 Systematically tracking composed use case occurrences

In Sect. 5.2.2, it was defined that the set of all active MSDs where, first, the MSDs belong to the same use case, and, second, all lifelines representing the same role are bound to the same objects, makes up the occurrence of a (non-composed) use case. The lifeline bindings imply which objects the roles of the use case specification are bound to. We have furthermore required that, if an active MSD is created, all lifelines are bound right away, i.e., there is

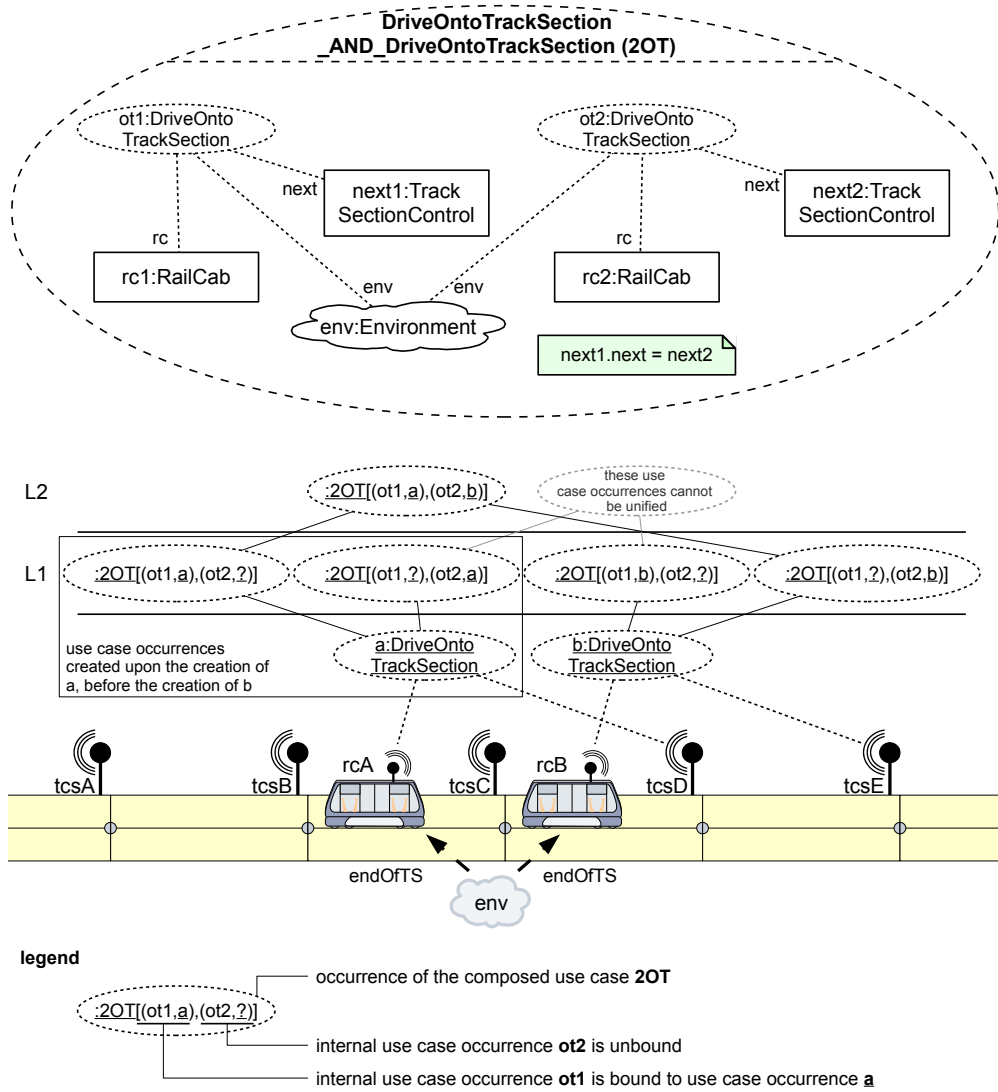


Figure 5.11: An example of incomplete and complete occurrences of a composed use case under consideration of a context expression

no late binding of lifelines. Thus, whenever a new active MSD is created or an active MSDs is terminated, use case occurrences may be created, the role bindings of use case occurrences may be updated, or use case occurrences may be terminated. Whenever this is the case, also the composed use case occurrences may be created, updated, or terminated. Instead of describing how to maintain the composed use case occurrences operationally, two rules are presented below that describe declaratively which composed use case occurrences must exist at any time during the simulation.

In the following, we distinguish certain *levels* of composed use case occurrences. Composed use case occurrences with just one bound internal use case are called *level-1* occurrences. Composed use case occurrences with two bound internal use case are called *level-2* occurrences, etc. Consequently, if a composed use case occurrence has n many internal use case occurrences, the *level- n* occurrences are such where all the internal use case occurrences are bound. (Examples of level-1 and level-2 occurrences of the composed use case 2OT were already shown in Fig. 5.10 and 5.11, in the rows labeled L1 and L2.) The composed use case occurrence of the different levels have to be created so that the following rules are satisfied.

Rule 1 (Level-1 occurrence of composed use cases). If an occurrence of a (non-composed or composed) use case exists, for each occurrence of this use case internally in a composed use case, an occurrence of that composed use case must exist where the internal use case occurrence binds the use case occurrence. The other internal use case occurrences remain unbound. Consequently, the roles of the composed use case occurrence are bound to the objects bound by the internal use case occurrence according to the role mappings that are specified in the collaboration of the composed use case. Only if a binding expression can be evaluated and evaluates to false, the according level-1 occurrence must not exist.

Rule 2 (Level-2.. n occurrence of composed use cases). Let $i \in 2..n$, where n is the total number of internal use case occurrences of a composed use case. For a set of level-1 use case occurrences, the following level- i occurrences of a composed use case must exist: For each level- $(i - 1)$ occurrence and for each unbound internal use case occurrence therein, there must exist a level- i use case occurrence for each level-1 use case occurrence that can be *unified* with the level- $(i - 1)$ occurrence. A level- $(i - 1)$ use case occurrence can be *unified*² with a level-1 use case occurrence if the bound internal use case occurrence in the level-1 use case occurrence is unbound in the level- $(i - 1)$ use case occurrence. Furthermore, each role mapped by that internal use case occurrence in the level- $(i - 1)$ use case occurrence must be either unbound or it must be bound to the same object as the according role in the level-1 use case occurrence. The *unified* level- i use case occurrence must be a copy of the level- $(i - 1)$ use case occurrence where additionally the internal use case occurrence that is bound in the unifiable level-1 use case occurrence is bound accordingly. The roles of

²Note that the unification of occurrences of composed use cases is a different concept than the unification of message events in MSDs, see Sect. 3.1.3

the composed use case occurrence must be bound to the objects bound by the internal use case occurrences according to the role mappings that are specified in the collaboration of the composed use case. Only if a binding expression can be evaluated and evaluate to false, the according level- i occurrence must not exist.

The play-out must be controlled according to all the complete and incomplete use case occurrences that are created according to the above rules. It may happen, however, that the configuration of cuts of the active MSDs that belong to an incomplete composed use case occurrence cannot be mapped to a state in the controller synthesized from the composed use case. That is because it may be that, for example, two use case occurrences that appear internally in a composed use case always occur simultaneously. In that case, there may be a situation where there exist level-1 composed use case occurrences where just one of the internal use case occurrences is bound, and the configuration of cuts of just the active MSDs within this one composed use case occurrence cannot be mapped to a state in the controller. In this case, the play-out is simply not controlled according to this incomplete composed use case occurrences.

5.3.6 Overly restrictive context expressions

One problem with the above approach is that *overly restrictive* context expressions in combination with a dynamic system may lead to simulation runs where use cases occur in a context that first does not satisfy the context expressions, but later the structure of the system changes such that the context expressions are satisfied. Then it may be that the play-out runs into violations that could have been avoided if weaker or no context expressions were specified. That is because the play-out of the active MSDs in the use case occurrences were not controlled right away according to the controller of the composed use case. Further examples will have to be studied in the future to identify such cases. In order not to track overly many composed use case occurrences during the simulation, which may make the approach inefficient, but in order to guide the play-out best as possible by the synthesized controllers, richer concepts may be required. One idea is to specify that different context expressions have to be satisfied in certain states of the composed use case occurrence.

5.4 Summary and Outlook

This chapter presented an extension to the play-out algorithm that allows it to avoid more avoidable violations during the play-out of an MSD specification by employing controllers that could be synthesized from parts of that specification. It was shown how single use cases can be specified in use case specifications, how active MSDs make up an occurrence of a use case during run-time, and how the execution of these MSDs can be controlled by a synthesized controller. These concepts were prototypically implemented in SCENARIOTOOLS. Furthermore, concepts were presented of how situations where use case occurrences overlap in a system can be captured in composed use case specifications, and how the

play-out can be guided by controllers that could be successfully synthesized from these composed use case specifications. It was also shown how it is envisioned that these techniques are applied in a scenario-based design process.

The concepts for guiding the play-out by controllers that could be successfully synthesized from composed use cases were not implemented in the scope of this thesis. It yet remains to be validated to which extent these concepts can be successfully used for simulating large systems with many overlapping use case occurrences.

In the future, it would furthermore be desirable to integrate not only controllers that could be successfully synthesized from use case specifications. If a use case specification is inconsistent, UPPAAL TIGA will synthesize a counter-strategy that contains the information of how the environment will always be able to violate the specification. In order to support the engineer in better understanding the nature of an inconsistency (after step 3 or 4 in the process shown in Fig. 5.1), it would be desirable to extend the SCENARIOTOOLS simulation so that it can also simulate the environment based on such counter-strategies. The engineer can then “play” against the environment in order to understand how the specification can be violated.

Triple Graph Grammar Extensions

The synthesis technique described in Chap. 4 is based on a mapping from MSD specifications, given in the form of stereotyped UML models, to networks of Timed Game Automata. This mapping is very complex, and it was therefore difficult to formalize the mapping and to build an automatic translator for it. The use of Triple Graph Grammars (TGGs) has greatly facilitated the specification and implementation of this mapping. It was possible to intuitively describe the correspondence between MSD and TGA constructs in the TGG rules.

However, some advanced TGG concepts and extensions to existing concepts and tools were necessary in order to successfully and conveniently apply TGGs for the task. The extensions to TGGs as previously introduced in Sect. 3.3 are explained in the following. First, an overview of the extensions is given in Sect. 6.1. The extensions have been implemented in a tool, called the TGG INTERPRETER, which is presented in Sect. 7.1.

6.1 Overview of the TGG extensions

Let us first overview the extended TGG concepts by the help of an example TGG rule. Figure 6.1 shows the TGG rule `MinimalEnvironmentMessage`. As the name suggests, it is a rule for mapping an environment message of an MSD that is the minimal (first) message of this MSD to certain constructs in the TGA. The diagram does not reveal the complete mapping represented by the rule, because the rule is a *specialization of* another, more general rule. The concept of TGG rule generalization is described by Klar et al. [KKS07]. In this thesis, these concepts have been improved and implemented in the TGG INTERPRETER. The dashed nodes in the rule diagram (1) are *refined nodes*, which are nodes that are originally defined by a more general rule and to which the specialized rule can add further edges or constraints. The concept of rule

generalization and the improvements that were elaborated in the scope of this thesis are explained in Sect. 6.2.

The TGG rule also contains *reusable nodes* and *reusable edges*, displayed gray with a “##” label (2a/2b). Reusable nodes/edges are such nodes/edges that can be interpreted either as context or produced nodes/edges. The purpose of these nodes/edges, their semantics, and how they are interpreted in a forward transformation scenario is explained in Sect. 6.5.

TGGs are furthermore extended with OCL constraints (3). OCL constraints can be used to specify the attribute values of objects that a node can be bound to. Also, OCL constraints can be used to formulate application conditions for the TGG rule. Section 6.3 explains further details. Last, the TGGs are extended to support *stereotype constraints* in UML domains (4). This is explained in Sect. 6.4.

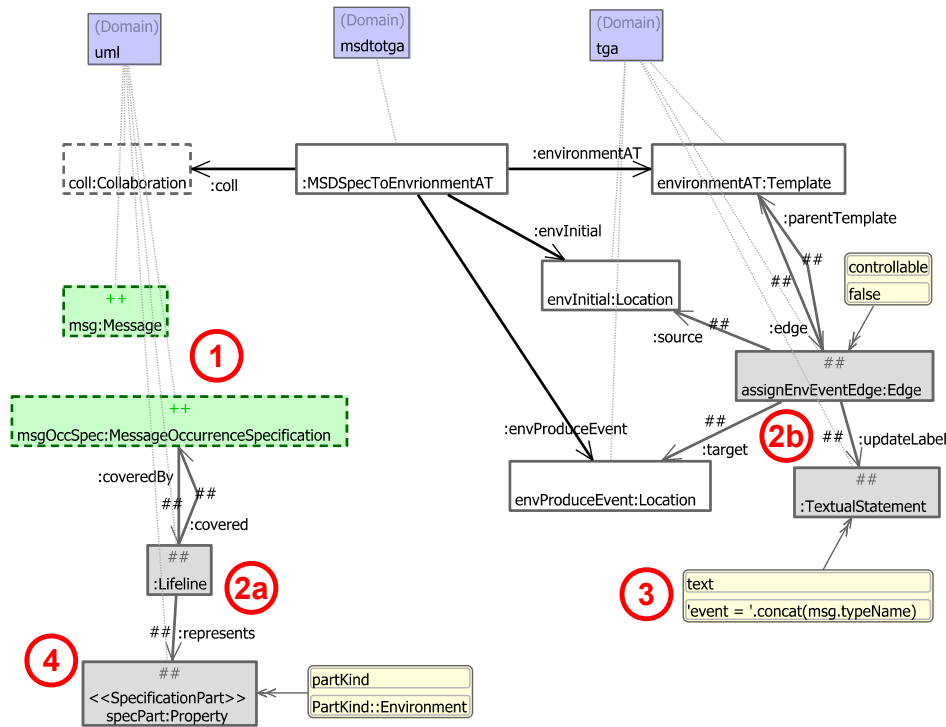


Figure 6.1: The TGG rule `MinimalEnvironmentMessage` illustrates extensions to the TGG formalism.

6.2 Generalization of TGG rules

Generalization, also called *inheritance*, is a powerful mechanism in object-oriented software engineering to reuse and extend existing solutions. Klar et al. have introduced this concept to TGG rules [KKS07]. They realized the concepts within the MOFLON tool suite.

6.2.1 Why a generalization concept for transformation rules?

In the MSD-to-TGA mapping, different kinds of messages need to be mapped to certain elements in the TGA system. Depending on the temperature of the message, its execution kind, or whether it appears as the first message in an MSD or not, different elements are created in the TGA system. Some elements in the TGA system are created for every message in an MSD specification, but others are only created in certain special cases. We find situations like these in many transformations. It may be that a certain pattern shall be transformed in a special way if for example objects in the pattern have a special type, special attribute values, or if these patterns are found in a special context.

In such cases, instead of explicitly specifying the complete TGG rules for every case, it is desirable to extract the commonality of certain rules in a more general rule, and then to only specify the parts that are special in addition to the general case in rules for the special cases. This way, we could avoid the tedious and error-prone task of specifying the common part of different TGG rules redundantly. Moreover, the TGG rule set would be much better maintainable, because if the common part of multiple TGG rules must be changed, we would only have to change it once in the more general rule, and not multiple times in the rules for the special cases.

This can be achieved with the TGG rule generalization concept that is presented in the following.

Figure 6.2 shows a generalization hierarchy of rules for mapping the different kinds of messages in an MSD specification. In the MSD-to-TGA mapping, we have to distinguish whether the message is a minimal (first) message in the MSD or a non-minimal message. If it is a minimal message, it must be distinguished if it is a system or environment message. For a non-minimal message it has to be distinguished if it is a hot or cold system or environment message. It must furthermore be distinguished between an executable and monitored (hot/cold and system/environment) message.

The mappings of all these different cases are similar. For example, for each message, an edge is created in the MSD automaton. This is specified in the most general rule, **Message**. Furthermore, the mappings for the minimal system or environment messages both add certain expressions to the edges in the MSD automaton. The mapping for these expressions is specified in the TGG rule **MinimalMessage**. Also the non-minimal hot and cold messages have commonalities, which are specified in the TGG rule **NonMinimalMessage**.

The TGG rule generalization relationships in Fig. 6.2 are represented by the closed arrow, inspired by the notation of generalizations in UML class diagrams. Some rules in this generalization hierarchy are *abstract*, indicated by the italic label text. This means that these rules shall not be applied directly—only non-abstract specializations of these rules can be applied directly. The semantics of generalized/specialized rules and abstract rules is explained in the following.

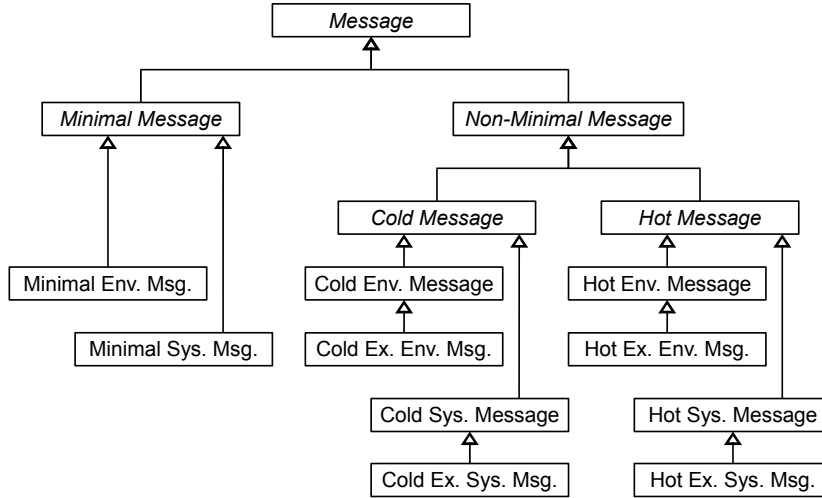


Figure 6.2: The generalization hierarchy of the TGG rules for translating different kinds of messages.

6.2.2 The TGG rule generalization concept by Klar et al.

A TGG rule describes a relation between a set of objects in one domain and a set of objects in another domain. Klar et al. argue that “generalization usually means that a member of a more specialized type also is a member of the more general type” and, therefore, whenever a more specialized TGG rule is applicable, also the more general rule should be applicable [KKS07, Sect. 4.1]. We call this the *guiding principle* of TGG rule generalization in the following.

To ensure that, Klar et al. declare a number of syntactical constraints for TGG rules that specialize other TGG rules. These constraints require, first (1), that a specialized rule contains a copy of the general rule [KKS07, rule 14]. Second (2), context nodes in the specialized rule may be replaced by nodes with a more specialized type [KKS07, rule 15]. Third (3), produced nodes in the general rule can be converted to context nodes in the specialized rule [KKS07, rule 15]. Forth (4), new nodes and edges may be added to the context and produced pattern of the rule [KKS07, rule 16], and, fifth (5), further attribute value constraints and application conditions may be added to the rule [KKS07, rule 17]. Last (6), the specialized rule must have a higher priority than the more general rule [KKS07, rule 10]. The priorities are considered in the transformation engine such that that a more general rule is only applied when any more specialized rule cannot be applied. This behavior is what we typically expect from a TGG transformation engine in a forward transformation scenario.

6.2.3 Improvements to the existing TGG rule generalization concept

The generalization approach taken in this thesis basically follows the above syntactical constraints. However, some changes and improvements to the generalization concepts of Klar et al. are introduced in this thesis.

Refined nodes

In the MOFLON tool suite, creating a specialized TGG rule quite literally requires to first create a copy of the more general rule and then to modify this rule according to the syntactical constraints. This, however, creates many redundancies in the TGG rules, which has a negative impact on their maintainability. Instead, an approach is presented in this thesis that allows the engineer to only specify the specific parts of a specialized rule inside a specialized rule—the general part of the rule is only specified in the more general rule.

The generalization mechanism developed here allows a TGG rule to express the generalization relationship to another rule by a reference to this rule. The copy of the general rule inside the specialized rule is then only created within the transformation engine during transformation-time, hidden from the user.

If extensions to the more general rule shall be added in the specialized rule, such as specializing the type of a node in the more general rule (see (2) above), adding an attribute constraint to a node in a more general rule (see (5) above), or adding edges from nodes in the more general rule to other, possibly new nodes (see (4) above), this can be done by the help of *refining nodes*. A refining node is a node in the specialized rule that represents a node in a more general rule. The latter node is also called a *refined node*. A refining node is displayed like a regular node, but has a dashed border (see Fig. 6.1). It is required that the refining node has the same name as the refined node, and that (according to (2) above) its type class is equal to or a subclass of the type class of the refined node (the original node in the general rule).

Klar et al. require that only context nodes in the specialized rule may have a specialized type (see (2) above). This seems overly restrictive, and it is not required by the guiding principle of TGG rule generalization; also the authors give no argument for this restriction. Therefore, this restriction is not included in this rule generalization approach.

The complete version of a specialized rule, i.e., the specializations and the copy of the more general rule, is created only during transformation-time, when the specialized rule is applied. A complete rule is formed by copying the contents of the more general rule into the specialized rule. Only the refined nodes of the more general rule are not copied. The edges in the more general rule that lead to or go out from refined nodes in the specialized rule are copied such that they lead to resp. go out from the according refining nodes instead. The same holds for attribute value constraints in the more general rule that are attached to refined nodes: in the complete copy of the specialized rule, these attribute constraints are attached to the according refining nodes. This process is applied recursively if the more general rule is a specialization of a yet more general rule.

Changing the “color” of a node in a specialized rule

Klar et al. propose that a produced node in a general rule may become a context node in the specialized rule (see (3) above). Intuitively, this seems to make the rule applicable in fewer cases. But, as the following example shows, there are examples where a more specialized rule would be applicable, but not the general

rule. This is a violation of the guiding principle of TGG rule generalization and is, therefore, not allowed in the TGG rule generalization approach presented here.

Consider the example shown in Fig. 6.3. On the left, two domain patterns (“columns”) of the same domain in two TGG rules are shown. The left pattern is that of a general rule and the right one is that of a more specialized rule. The node $b:B$ is a produced node in the general rule and becomes a context node in the specialized rule. Now imagine that the shown domains are the source domains in a forward transformation scenario and there is a situation in the source domain instance model where there are the instances $:A$, $:B$, $:C$, and where $:A$ and $:B$ are already bound (i.e., already matched by another rule, indicated by the check symbol). According to the forward transformation interpretation of a TGG rule as explained in Sect. 3.3, Fig. 3.18, the specialized rule’s source pattern is applicable, because the context pattern can be matched to already bound nodes and the produced pattern can be matched to unbound nodes. However, the source domain pattern of the more general rule is not applicable in this case: the context pattern ($a:A$) can be matched to a bound node, but the produced pattern cannot be matched because the object $b:B$ is already bound. Thus, this violates the guiding principle of TGG rule generalization.

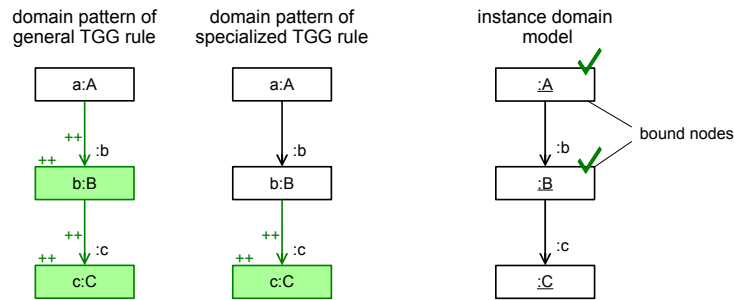


Figure 6.3: An example where a specialized rule (according to Klar et al.) is applicable, but not the general rule.

Therefore, the generalization approach presented here requires that produced nodes and context nodes must “preserve their color”, i.e., if the refined node is a produced node or a context node, the refining node must be a produced node or a context node, respectively. An exception are the reusable nodes. As will be explained shortly (Sect. 6.5), reusable nodes can be interpreted as either a produced node or a context node. In a more specialized rule, it may be decided to turn the refining node of a reusable node into a produced or context node. In the course of this thesis, there was however no case encountered where this was useful or necessary.

Precedence instead of priority for specialized rules

In a TGG, all applicable rules can be applied non-deterministically. In principle, this also holds if there are generalization relationships among the TGG rules.

In a forward transformation scenario, however, the desired behavior is typically, that a rule is only applied if not a more special rule can be applied.

Klar et al. require that the specialized rules shall have a higher *priority* than the more general rules (see (6) above). Operationally, the MOFLON TGG engine will always try to apply a rule with a higher priority before trying to applying a rule with a lower priority. In the approach presented by Klar et al., the priorities must be explicitly assigned to the rules.

The approach presented here does not rely on priorities. Instead the generalization relationships imply *precedences* among the TGG rules. A specialized rule always has *precedence over* a more general rule. Similar to the priority, the precedence ensures that a rule will not be applied if a rule with precedence over this rule is applicable. Operationally, the TGG INTERPRETER will only try to apply a rule if it was not possible to apply rules with precedence over this rule. This mechanism also ensures that never a more general rule is applied if a more specialized rule is applicable. However, there are two advantages with this approach compared to the approach with the priorities. First, the transformation engineer will not have to explicitly assign any priority to the rules. Secondly, the transformation engine can, after unsuccessfully trying to apply all specializations of a more general rule, try to apply the more general rule right away. In the approach using the priorities, the transformation engine may be required to first try to apply other rules if there are rules that it did not try to apply yet and that have a higher priority. This may have a disadvantage concerning the efficiency as illustrated by the following example.

As an example consider the generalization hierarchy shown in Fig. 6.2. The transformation engine may first try to apply the rule `ColdExecutedSystemMessage` and, if the rule cannot successfully be applied to the model, the engine may try the rule `ColdSystemMessage` next. This is different from the approach proposed by Klar et al. If we assume that the priority of the TGG rule `Message` is 0 and increases by one with each specialization, then the rules `ColdExecutedSystemMessage` and `HotExecutedSystemMessage` have the priority 4 and the TGG rule `ColdSystemMessage` has the priority 3. Therefore, the transformation engine would have to try to apply the rule `HotExecutedSystemMessage` before trying to apply the rule `ColdSystemMessage`.

The advantage of the precedence-approach is that it leaves more room, for example, to apply heuristics that can improve the speed of the transformation engine. Such a heuristic may for example suggest rules that are more likely to be successfully applicable, based on the currently unbound source elements and the source produced pattern of the rule. For example, when there is an unbound cold message in the model, but the rule `ColdExecutedSystemMessage` is not applicable, it makes sense to first try to apply the TGG rule `ColdSystemMessage`, instead of trying to apply `HotExecutedSystemMessage`.

6.3 OCL attribute value constraints

Within this thesis, TGGs were extended with attribute value constraints that can be specified with OCL [OCL10]. This extension is implemented within the TGG INTERPRETER.

OCL is particularly convenient for formulating conditions and queries in object-oriented models. OCL is for example used in QVT for describing model patterns and attribute constraints [QVT08]. Powerful interpreters for OCL queries have been developed in the past, for example in the Eclipse Modeling Project¹, which is also integrated by the TGG INTERPRETER.

Figure 6.4 shows one constraint from the TGG rule `MinimalEnvironment-Message` as shown in Fig. 6.1, and it highlights the terminology that is used in the following. First, an attribute value constraint is always attached to a node, which is called the constraint's *slot node*. Second, the top row of the rounded rectangle of the attribute value constraint shows the attribute that this constraint requires a certain value for. This attribute is called the constraint's *slot attribute*. Last, the bottom row of the constraint's rectangle contains the *value expression*, which specifies the value required for the attribute. An object can only be bound to the slot node if the value for the slot attribute equals the evaluation of the value expression.

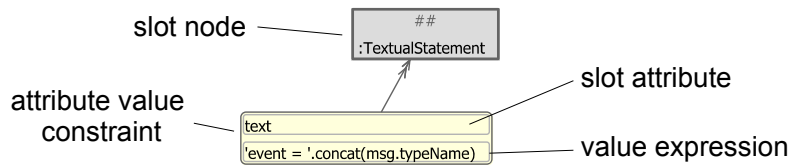


Figure 6.4: An *attribute value constraint* is attached to a *slot node* and constrains the value of the *slot attribute* by the value specified by the *value expression*

OCL is integrated into the TGGs such that nodes, if they are named, can be used as variables in the constraints. The value expression of the attribute value constraint shown in Fig. 6.4 for example contains the variable `msg`, which is a node in the TGG rule. (Actually it is a node in the more general rule, which appears in the diagram as a refining node just to make it better visible where the variable `msg` comes from.) These variables are also called *node variables*. Here the expression specifies a string, which is the concatenation of the string `'event = '` and value of the attribute `typeName` of the object bound by the node `msg`. Also the variable `self` may be used in the value expression; `self` is bound to the object bound by the slot node of the attribute value constraint.

The value expression of an attribute value constraint is evaluated immediately when the transformation engine tries to match the node that the attribute value constraint is attached to or when an object is created according to the respective node in the target or correspondence model. It may be, however, that a value expression cannot be evaluated right away, because one of the nodes

¹<http://www.eclipse.org/modeling/mdt/?project=ocl#ocl>

that occur in the value expression as a variable is not yet bound. Then the expression is evaluated as soon as all the nodes that appear in the expression as variables are bound. The latter case is called the *delayed evaluation* of the value expression in contrast to the *immediate evaluation*.

A node in the source or context pattern of a rule can be matched if for each attribute value constraint attached to the node, the object's value for the slot attribute equals the evaluation of the value expression. The graph matching algorithm in the TGG INTERPRETER backtracks if a node cannot be matched due to an immediate evaluation of an attribute constraint, i.e., it tries to find another valid match for that node. When it turns out that a node is not a valid match for an object upon a delayed evaluation, then the interpreter first tries to find another match for the node that it matched last. An alternative match for that node means another binding for a variable in the value expression, by which the expression may render the required value. Only if all the candidates that the interpreter tries for that last node do not satisfy the attribute value constraint, it backtracks further. Eventually it may try to match the slot node of the unsatisfied attribute value constraint to another object.

For an object created in the target and correspondence model the value of the value expression is assigned to the slot attribute as soon as it can be evaluated.

In addition to attribute value constraints, a TGG rule may contain *application conditions*. Application conditions are represented by the same rounded rectangles used for attribute value constraints, but they do not specify any slot attribute and do not necessarily need to specify a slot node. Moreover, the value expression of an application must evaluate to a Boolean value. If no slot node is specified, the value expression is evaluated as soon as bindings are available for all the node variables. The transformation engine backtracks to find satisfactory matches for the nodes and the rule is not applicable when no satisfactory matches for the nodes can be found. One can specify a slot node for application conditions. This way, the variable `self` can be used to represent the slot node in the value expression. Also, the TGG INTERPRETER will not evaluate the attribute value constraint before it tries to match the slot node. This may yield a slight performance increase, because the TGG INTERPRETER will not try to evaluate the condition before there is a binding candidate for a node that is involved in the application condition. Specifying a slot node or not does however not change the semantics of an application condition or the rule.

The implementation of the TGG INTERPRETER also allows the transformation engineer to define a number of custom OCL operations and attributes in an external text file. The MSD-to-TGA transformation heavily uses such operations and attributes, see Sect. B.2. The value expression of the attribute value constraint shown in Fig. 6.4 for example accesses the property `typedName` of the variable `msg`. This property is, however, no property of the UML meta-class `Message`, but it is a custom attribute definition specified in OCL, see the Listing B.1 in Sect. B.2.

Dang and Gogolla [DG08, Dan09] presented a similar approach for specifying attribute value constraints and application conditions within TGGs. In their approach, they specify TGG rules textually, including a number of OCL statements. Then a framework called USE can operationalize the TGG rules,

including the evaluation of OCL constraints and the execution of assignments on objects that are created on the target side. Compared to the approach presented here, however, it is not possible to define custom OCL operations and attributes in an external text file. Also, they do not support many advanced TGG features, such as generalization or support for TGGs that specify a mapping between more than two domains. Wagner [Wag09] also describes how to incorporate OCL into TGGs, but this extension was not elaborated in detail, nor implemented in the tool (FUJABA).

6.4 UML stereotype constraints

UML profiles are a powerful mechanism to extend UML for specific purposes. A profile can define stereotypes that can be applied to certain UML elements, and these stereotypes can also define a number of additional properties for the stereotyped UML element. Profiles are used in this thesis to extend UML sequence diagrams to Modal Sequence Diagrams (see Sect. A.2). A stereotype for messages, for example, adds the attributes *temperature* and *execution kind* to messages.

To formulate constraints on whether a certain stereotype is applied to a UML element, TGGs were extended by *stereotype constraints* in the scope of this thesis, which can be specified for nodes in a UML domain.

The TGG rule `MinimalEnvironmentMessage` shown in Fig. 6.1 has a stereotype constraint applied to the node `specPart:Property`. which is indicated by the additional label `«SpecificationPart»`. Such a stereotype constraint expresses that a node can only match the UML element object when the specified stereotype, here `SpecificationPart`, is applied to that element. Nodes can also have multiple stereotype constraints. Then all of the specified stereotypes must be applied to the UML element. Furthermore, it is possible to specify negative stereotype constraints. Then the stereotype specification label in the node has the form `not «...»`. See the TGG rule `Assignment` shown in Fig. B.30 for an example. A negative stereotype constraint means that the node can only match a UML element that does not have this stereotype applied.

Furthermore, if a node has a stereotype constraint, attribute value constraints can be attached to the node that refer to a slot attribute defined by the stereotype that the stereotype constraint requires to be applied. If such an attribute value constraint is attached to a node, the node can only match an object if the following conditions hold. First, as explained above, the stereotype constraint of course requires that the object is a UML element with the respective stereotype applied. Second, it is required that the application of the stereotype carries a value for the slot attribute that equals the evaluation of the attribute value constraint's value expression.

6.5 Reusable nodes

According to the TGG semantics introduced in Sect. 3.3, an object in the model can be bound exactly once by a produced node. But there are often cases where

the transformation engineer wishes to specify a mapping where an object may or may not yet have been bound (i.e., matched or created) by another rule application. Then the transformation engineer would have to specify two rules: one where the respective node is a context node and another rule where the respective node is a produced node. This may lead to many redundant rules.

Instead the concept of *reusable nodes* [Gre06, GK10] was introduced, which allow the transformation engineer to subsume these cases in one rule (see also Kindler and Wagner [KW07]). Reusable nodes are represented in TGG rules as gray nodes that have a “##” label. They can be bound to already bound or yet unbound objects. In addition to reusable nodes, there are also *reusable edges*. Reusable edges are also colored gray and have a “##” label. Similar to reusable nodes, reusable edges can be bound to already bound or yet unbound links in the model. For simplicity, the following explanations focus on the reusable nodes, but same the principles apply to reusable edges as well.

Operationally interpreting reusable nodes in a source domain during a forward transformation is easy: we just match an object without considering whether the object is already bound by a previous rule application (this implies that we interpret the reusable node as a context node) or not (this implies that we interpret the reusable node as a produced node).

In the target domain, the transformation engine can similarly choose to either create an according object anew or to match a previously created (and therefore already bound) object. The latter choice is also called *reusing* an object. In order to control these choices, application conditions can be used [GK10]. Mostly, however, a reasonable default behavior for a transformation engine is to reuse an existing object where it is possible and to only create an object anew where it is not possible. The specification of the declarative QVT languages specifies a similar behavior, called the *check-before-enforce semantics* [QVT08].

Driven by the specific requirements in the MSD-to-TGA mapping, a specific operational semantics for the interpretation of reusable nodes on the target side during a forward transformation was elaborated in this thesis and implemented in the TGG INTERPRETER.

6.5.1 Reusable nodes in the example

An example of a rule that contains reusable nodes is the TGG rule `Minimal-EnvironmentMessage` shown above in Fig. 6.1. The rule maps environment messages in the MSD specification to different elements in the TGA model. It contains reusable nodes in the UML domain (source domain) as well as in the TGA domain (target domain). In the UML domain, the nodes `:Lifeline` and `specPart:Property` are reusable, in the TGA domain, the nodes `assignEnvEvent-Edge:Edge` and `:TextualStatement` are reusable.

Reusable nodes in the source domain

A message is an environment message when the sending lifeline represents an environment object. In the UML model, this information is represented by the

attribute `partKind` of the stereotype `SpecificationPart`, which is attached to the role in the collaboration that represents the object. Therefore, the nodes `:Lifeline` and `specPart:Property` occur in the TGG rule `MinimalEnvironmentMessage`, including the stereotype constraint and the attribute value constraint that checks the value of this attribute. Before a rule for translating a message in an MSD is applied for the first time, the sending lifeline and the role (property) represented by that lifeline are unbound. Then, when further rules are applied for translating messages, the role (property) or also the sending lifeline may already be bound. Instead of specifying multiple rules where the nodes `:Lifeline` and `specPart:Property` occur in different combinations as context or produced nodes, there is only one rule where they occur as reusable nodes.

Reusable nodes in the target domain

In the target domain, the rule `MinimalEnvironmentMessage` states, among other things, that an edge going from the location `environmentInitial` to the location `produceEvent` in the environment template automaton, assigning a value to the variable `event`, shall be created or reused. In this case, the intended behavior is that it shall be created anew if not yet any message representing the same event has been translated previously, otherwise this edge shall be reused.

Figure 6.5 illustrates an example where there are three MSDs in which the message `m1` is sent from the lifeline `env:Env` (representing an environment object) to a lifeline `a:A`, representing a system object. All these messages shall be translated (among other things) to the same edge in the environment automaton. In the first MSD, the message is the minimal message and is mapped by the TGG rule `MinimalEnvironmentMessage`. In the second MSD, there are two messages which represent the same event. One is again the minimal message and is mapped by the TGG rule `MinimalEnvironmentMessage`. The other is a non-minimal cold message and is therefore mapped by the TGG rule `ColdEnvironmentMessage`. In the third MSD, probably an assumption MSD, the message appears as a non-minimal hot executed message and is translated by the TGG rule `HotExecutedEnvironmentMessage`. Whichever of these rule applications takes place first creates the edge and the attached update label. The desired behavior is then that all other rule applications reuse this edge and its update label.

6.5.2 The operational semantics of reusable nodes in the target domain during a forward transformation

As mentioned above, in a forward transformation scenario, the transformation engine could choose to match a reusable node on the target side to an existing object, i.e., to reuse that object, or to create an according object anew. The intention of the above example rule is to always reuse a certain edge object with a certain update label in the target model when it already exists. Therefore, mechanisms are required to ensure the reuse of objects in certain cases.

One possibility to control the reuse of objects is by using attribute value constraints and application conditions. In an application condition, we could

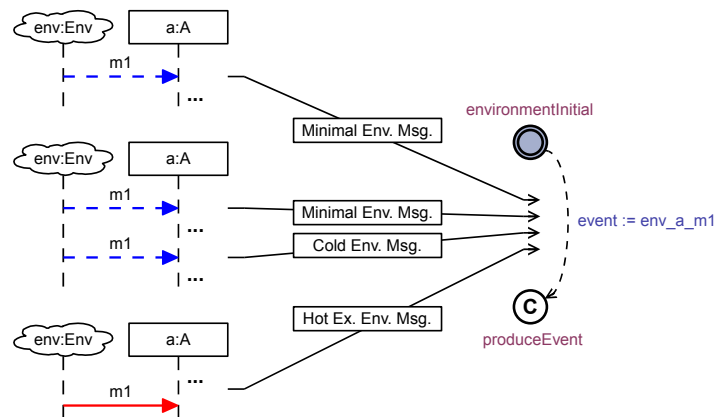


Figure 6.5: An illustration of how many messages representing the same event are mapped to the same edge by different TGG rules

for example express that there must not be two edges between the same locations with the same guard, update, and synchronization label. Instead of specifying these application conditions for each rule, which could lead to redundancies, we could also specify constraints of the models of the involved domains once per transformation. These must then hold at the end of a transformation. We call such conditions *global constraints* [GK10].

However, it would be inefficient for the transformation engine to first try to create new objects, to then find out that this violates certain application conditions, and then to try to reuse already existing objects. Therefore, it is reasonable for the transformation engine in a forward transformation scenario to first try to reuse existing objects and to create objects anew only if that is not possible.

But should the transformation engine make the decision to reuse an object separately for each node? Let us again consider the reusable nodes in the TGG rule `MinimalEnvironmentMessage` (Fig. 6.1). Here there are two reusable nodes in the target domain, one representing an edge object, one representing an update label statement. Consider a case where this rule is applied to translate a certain environment message, but some environment messages in an MSD specification were already translated to their corresponding TGA model elements previously. What could happen now, if the decision to reuse an object is made separately for each node, is that the transformation engine first checks whether there is an edge object that can be reused for the node `assignEnvEventEdge:Edge`. Because there were previously already some edge objects created, and no particular attribute value constraints are attached to the node, the transformation engine will choose one for reuse. Next, it will try to reuse the attached update label statement for the reusable node `:TextualStatement`. Let us assume that the edge object that was reused has no attached update label that satisfies the attribute value constraint attached to the node `:TextualStatement`. Therefore, the transformation engine would now create a new update label statement for

the edge. The result will be an edge with possibly two update label statements, both assigning another value to the variable `event`.

The outcome of such a transformation is clearly not what we intended in this case. In order to control the reuse of objects for the `assignEnvEventEdge:Edge` node, we could add further application conditions, but this seems very tedious. Instead, we could require that existing objects in the target domain should only be reused if we can find objects to be reused for each reusable node in the target pattern of a rule. QVT's check-before-enforce semantics formulates a similar rule [QVT08]. We call this the *reuse-all-or-create-all directive* in the following. But would that solution be satisfactory?

Let us again consider the TGG rule `MinimalEnvironmentMessage` (Fig. 6.1). In the target domain, we find the reusable nodes `assignEnvEventEdge:Edge` and `:TextualStatement`. Furthermore, however, the rule indirectly inherits from the abstract rule `Message`. Among other things, it inherits the reusable node `:BooleanTextualExpression` from this rule (see Fig. B.13, p. 215). This node represents a fragment in a return expression of the function `bool eventInMSD(int ev)` that is created in the MSD automaton template corresponding to the MSD that is the parent of the message (see Listing 4.3). This expression fragment shall be created or reused when a message is translated.

Consider the case where we translate an MSD specification where there are two MSDs in which the same minimal environment message appears, i.e., there are two MSDs with first messages that refer to the same operation and that are sent from and to lifelines that represent the same objects/roles. Both of these messages will be translated by the `MinimalEnvironmentMessage` rule. During the translation of the first message, a new edge will be created between the location `environmentInitial` and the location `produceEvent`. Furthermore, an according expression fragment will be added in the return statement of the function `bool eventInMSD(int ev)` that is created in the MSD automaton template that corresponds to the parent MSD of the message. During the translation of the other message in the second MSD, the transformation engine would then consider the reuse of the edge that has previously been created between the location `environmentInitial` and the location `produceEvent`. However, in the function `bool eventInMSD(int ev)` of the MSD automaton template that corresponds to the second MSD, no expression fragment was yet created that can be reused by the node `:TextualStatement`. According to the reuse-all-or-create-all directive, we would then reuse no objects, but instead create objects for all reusable nodes. This leads to two redundant edges in the environment automaton, which is not the desired behavior in this case.

A solution to the problem is that the reuse-all-or-create-all directive does not hold for all reusable nodes in a TGG rule, but only for certain groups of reusable nodes in the rule. We call such a group a *reusable pattern*. In the current implementation of the TGG INTERPRETER, a reusable pattern is such a pattern in a TGG rule where reusable nodes and edges form a maximal connected subgraph of the rule graph. So, for example, the reusable nodes `assignEnvEventEdge:Edge` and `:TextualStatement` (and the edge between them) forms one reusable pattern. The reusable node `:BooleanTextualExpression` forms

another reusable pattern, because it is not connected to the other reusable pattern via other reusable nodes and edges.

In the future, it may be desirable to not just imply that a reusable pattern is formed where reusable nodes and edges form a maximal subgraph. It may be that in such a pattern there are actually multiple patterns for which the transformation engineer desires the reuse-all-or-create-all directive to hold separately. Therefore, it should be possible for the transformation engineer to specify explicitly which sets of nodes form a reusable pattern.

The reuse-all-or-create-all directive must hold for reusable patterns in all domains and also in other transformation scenarios. The TGG INTERPRETER is implemented such that there is a priority on the reuse of objects in the target pattern during a forward transformation. However, the priority on the reuse is not part of the TGG semantics in general. There may be other examples and other application scenarios where this priority is not practical.

Reusable patterns are an important and relevant concept that have not been proposed for TGGs previously.

6.6 Summary

Within the scope of this thesis, TGGs were extended by a number of vital concepts for realizing complex transformations such as the MSD-to-TGA transformation (see Appendix B). The extensions include a generalization concept of TGG rules which improves the concepts presented previously by Klar et al.[KKS07]. In the MSD-to-TGA mapping the generalization greatly reduces the number of rules and the redundancy among the rules compared to a TGG without generalization. Another extension to TGGs are the reusable nodes, which also reduce the number of TGG rules when there are cases in a transformation where the objects in the source and target models may or may not have been already matched resp. created. TGGs were also extended such that OCL expressions can be used in attribute value constraints and application conditions. Last, constraints can be added in UML domains to specify the application of stereotypes and to specify attribute value constraints over stereotype attributes.

6.7 Outlook

In the future, it is desirable to investigate further concepts for reuse of TGG rules. It can be observed in the MSD-to-TGA transformation that, despite the extensive use of rule generalizations, there are still redundancies in the rules. For example, there are four rules for different kinds of executed messages (`ColdExecutedEnvironmentMessage`, `ColdExecutedSystemMessage`, `HotExecutedEnvironmentMessage`, and `HotExecutedSystemMessage`, see Fig. 6.2). The patterns that these rules “add” to their more general rule are all very similar (see Fig. B.21, B.23, B.26, and B.28).

Furthermore, it was explained in Sect. 5.3.2 that we would like to also translate composed use cases to a system of TGAs so that a controller strategy can be synthesized for the composed use cases. If a composed use case describes

the occurrence of use cases where two or more occurrences are occurrences of the same use case, multiple MSD automaton templates, one for each internal use case occurrence, will have to be created for the same MSDs in the use case specification of the internal use case occurrence. Figure 6.6 illustrates this.

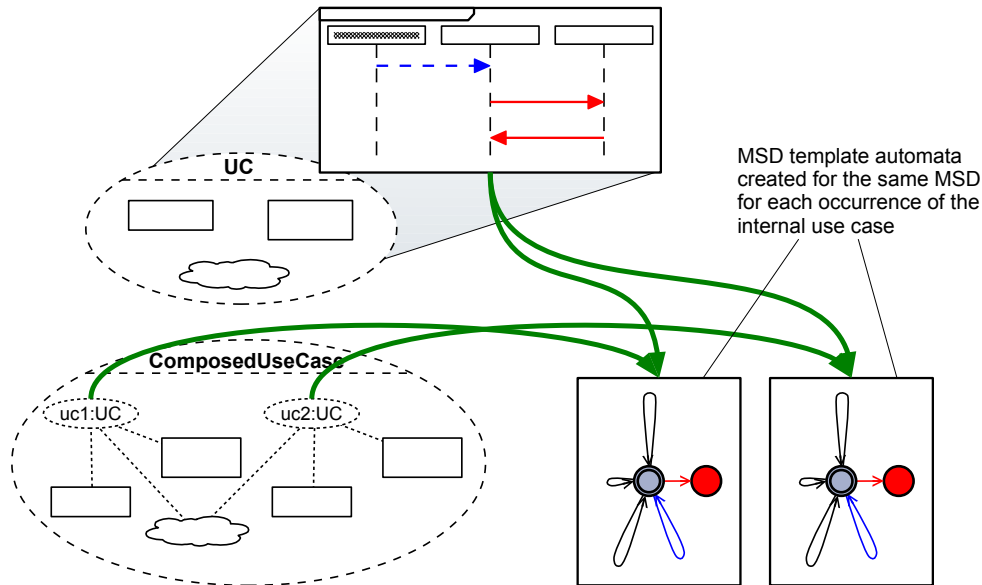


Figure 6.6: In the translation of composed use cases to a TGA network, the same MSD may have to be mapped to multiple MSD template automata if the use case occurs as an internal use case occurrence in the composed use case several times

The problem in creating these copies is that TGG rules must match the elements in the MSD multiple times to create the corresponding elements in multiple MSD template automata. However, the bind-only-once semantics of the produced nodes allows an object to be matched only once by the produced source pattern of a TGG rule. It seems that an additional TGG concept will be required to solve this problems. It may be possible to define that the bind-only-once directive must only hold within a certain scope of rule applications.

Realization and Evaluation

The major concepts presented in this thesis were realized in a number of software tools. This chapter presents the TGG INTERPRETER in Sect. 7.1. The synthesis of controllers from MSD specifications is realized in a tool chain as described in Sect. 7.2. Editors for modeling use case specifications and a play-out algorithm that can be guided by controllers synthesized from use case specifications were implemented within the SCENARIOTOOLS tool suite, which is presented in Sect. 7.3. This chapter last summarizes evaluation results obtained with these tools in Sect. 7.4.

7.1 The TGG Interpreter

The TGG INTERPRETER is a tool for model-to-model (M2M) transformations in ECLIPSE¹. The TGG INTERPRETER can transform models of the Eclipse Modeling Framework (EMF)². In addition to a transformation engine, the tool includes a graphical TGG rule editor, and basic UI support for configuring and executing transformations in the ECLIPSE workbench. A first version of the TGG INTERPRETER was implemented in the scope of the author's master thesis [Gre06], but all components were redesigned in the scope of this thesis in order to incorporate new features (see Chap. 6), and to increase the performance and extensibility.

Figure 7.1 shows a screenshot of the graphical TGG rule editor. The rule currently edited is the rule `MinimalEnvironmentMessage` as shown in Fig. 6.1. The left of the screenshot shows the palette of the TGG rule editor, where tools can be selected to create new nodes, edges, and constraints. The editor is *type-model-sensitive*, which means that it aids the user in creating only graph

¹<http://www.eclipse.org/>

²<http://www.eclipse.org/emf/>

patterns that represent valid model structures according to the type model of a particular domain. The bottom shows the properties view, which allows the user to modify the properties of the nodes, edges, and constraints in the diagram. Currently, the properties of the selected constraint node are shown.

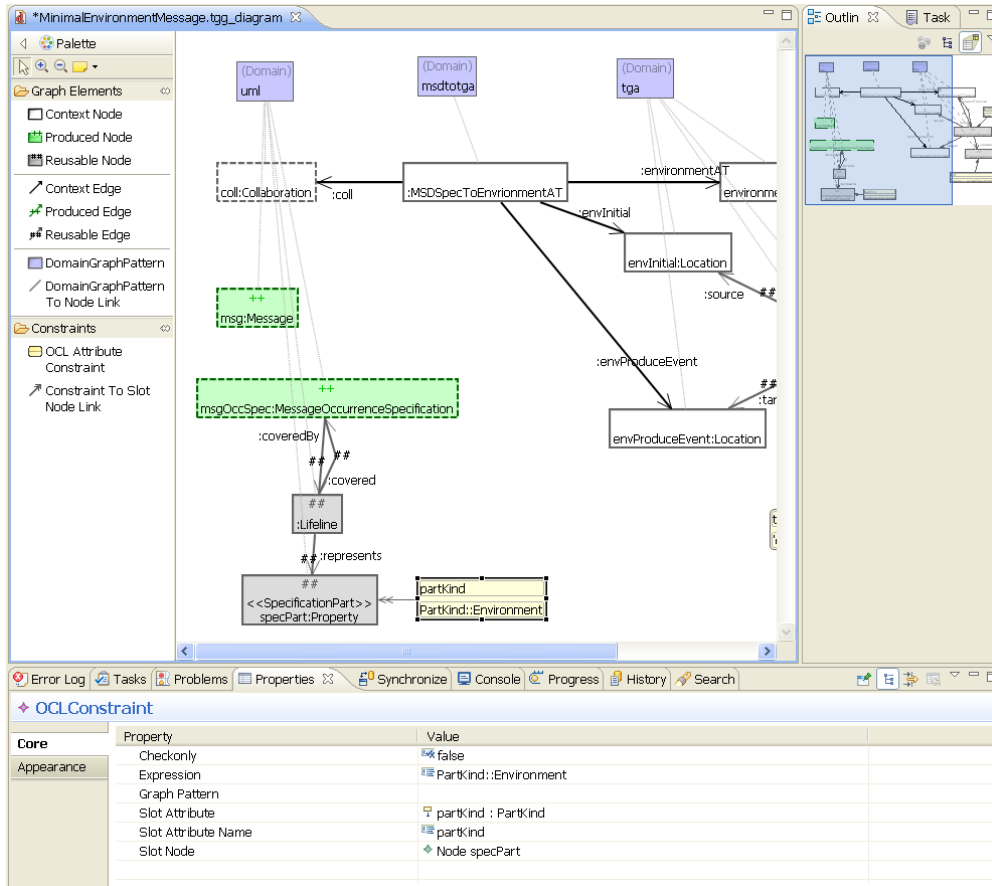


Figure 7.1: A screenshot of the TGG rule editor

TGG transformations can be configured and executed by simple actions that are integrated in the ECLIPSE user interface. A transformation can be configured and stored in an interpreter configuration model. This model contains the information of which source model is to be transformed by which TGG, and where the resulting target model shall be stored. Figure 7.2 shows a screenshot of how the user can select the command to perform TGG transformation. For more information about the TGG INTERPRETER, visit the TGG INTERPRETER website³.

³<http://www.cs.uni-paderborn.de/en/research-group/software-engineering/research/projects/tgg-interpreter.html>

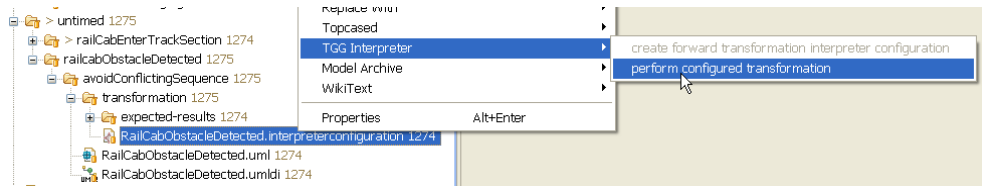


Figure 7.2: A transformation can be executed by a simple right-click action on an interpreter configuration file

7.2 MSD-to-TGA mapping and synthesis

The mapping of the MSD specification to a network of TGA is realized by a two-step transformation approach. First, the MSD specification UML model is transformed into an EMF model of a TGA network by a TGG transformation (see the transformation specification in Appendix B). Then this model is transformed to a valid input file for UPPAAL TIGA via an XPAND⁴ model-to-text (M2T) transformation. In combination with a winning condition (see Sect. 4.4), UPPAAL TIGA can then synthesize winning strategies for the TGA system if the MSD specification is consistent. Otherwise, it will produce a counter-strategy, which is a strategy that describes how it will be possible for the environment to violate the winning condition resp. the MSD specification. Figure 7.3 illustrates this process.

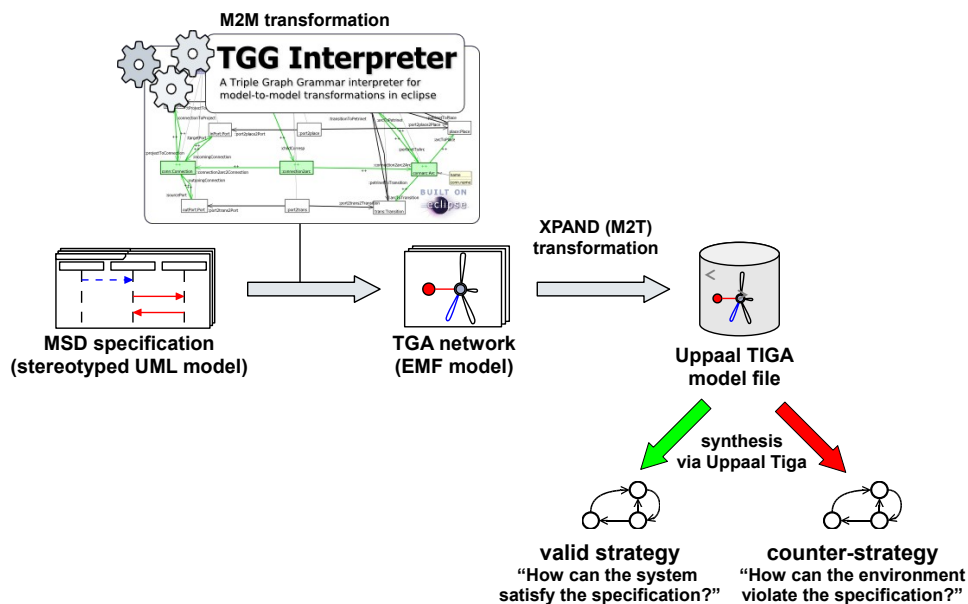


Figure 7.3: The two-step transformation approach

⁴see <http://wiki.eclipse.org/Xpand>

Between the TGG and XPAND transformations, there are two further steps involved that are not shown in the figure. First, a simple Java component sorts the function declarations in the EMF model of the TGA system. This is necessary, because in UPPAAL, functions can only call functions declared prior to their own declaration. The Java component makes sure that the order of the function declarations complies with this constraint. Such an order cannot currently be ensured by the TGG transformation. Second, the negation of time conditions expressions (replacing the operator $<$ by \geq , or \leq by $>$, and vice versa) is carried out by another small Java component, since this cannot be conveniently achieved in TGGs or OCL. These two Java components and the XPAND transformation are assembled in an oAW (openArchitectureWare) workflow⁵ that can be executed in ECLIPSE.

7.3 ScenarioTools

SCENARIOTOOLS integrates editors for specifying use case specifications, the above-mentioned synthesis tool chain, and an implementation of the play-out algorithm that can be guided by controllers synthesized from use case specifications. Major parts of SCENARIOTOOLS were realized by a *project group*, a two-semester course for master students that took place at the software engineering group of Prof. Schäfer in 2009 and 2010.

7.3.1 Modeling

A scenario-based specification of a system consists of a number of use case specifications. As described in Sect. 4.2, a use case specification consists of a class model, a collaboration diagram, and a number of MSDs. For modeling these different parts of a use case specification, SCENARIOTOOLS relies on the TOPCASED⁶ UML editors.

Modeling MSDs

MSDs can be modeled in UML with an additional profile that, for example, extends messages in sequence diagrams with attributes for their temperature and an execution kind. The profile used by SCENARIOTOOLS is an extension of the profile defined by Maoz and Harel [MH06, HM08]. Section A.2 contains a detailed documentation of the profile. In order to graphically show the temperature and the execution kind of messages, the TOPCASED sequence diagram editor component was extended and modified. Figure 7.8 shows a screenshot of the editor.

⁵see [http://wiki.eclipse.org/Modeling_Workflow_Engine_\(MWE\)](http://wiki.eclipse.org/Modeling_Workflow_Engine_(MWE))

⁶<http://www.topcased.org/>

Package merge for a flexible combination and reuse of use case specifications

The collaboration and sequence diagrams in the use case specifications can be based on one class model; alternatively, SCENARIOTOOLS allows the engineers to specify *separate* class models for the use case specifications that can later be integrated in one class model via *package merge*. Package merge is a mechanism defined by UML [UML09, Sect. 7.3.40, pp. 112] that describes how the classes, properties, associations, etc. defined in one or several class diagrams can be merged with the contents of another class diagram. This allows the engineer to keep extensions to the class model that are made for the specification of different use cases separate from each other. This has the advantage that it is more easily traceable which classes, properties, operations, and associations were added to the class model for which use case specification. Furthermore, this way SCENARIOTOOLS allows the engineer to *flexibly combine* different use case specifications, for example to *define certain variants* of a product, or to *reuse use case specifications in other contexts*.

Instead of creating one big class model, SCENARIOTOOLS for example supports the following modeling process where use case-specific extensions to the class model are captured in separate packages: Imagine that the engineers start specifying the system by providing a first class model of the system, which we call *base model* in the following. For example, the engineer could specify a base model for the RailCab system that describes that a RailCab system consists of RailCabs that have a certain speed and weight, may be registered to a current track section control, and react to events in their environment. Track section controls can have previous and next track section controls. The top of Fig. 7.4 shows such a class diagram (which is of course just a simplified model of a RailCab system).

Then, when specifying the use case Drive onto track section (see Fig. 3.1 or 4.2), there are no new classes added to the base model, but there are new types of messages introduced that can be received by the RailCabs and track section controls. These message types are represented in the class model as operations. Instead of adding these operations to the formerly created classes, the engineers can create a new class model, define the specific extensions made to the first class model in this new class model, and later *merge* the base class model with these use case-specific extensions. The extensions to the base class model that are specific for the use case Drive onto track section are shown in the package in the middle of Fig. 7.4. This package has a merge relationship to the base package, which means that it consists of the contents of the merged package RailCabBase, but, in this case, adds operations to the classes RailCab and TrackSectionControl.

Imagine further that the engineers now specify the use case of a RailCab driving onto a merging switch. One scenario in this use case is that the RailCab must not be given the permission to enter a merging switch too early (see Fig. 3.8). In this use case, messages appear that refer to operations that were added in the package of the use case specification Drive onto track section. But, furthermore, the class MergingSwitchControl is added to the class model in this

use case specification. Again, instead of adding this class to the package of the use case specification Drive onto track section, this extension is modeled in a separate package, see the bottom of Fig. 7.4.

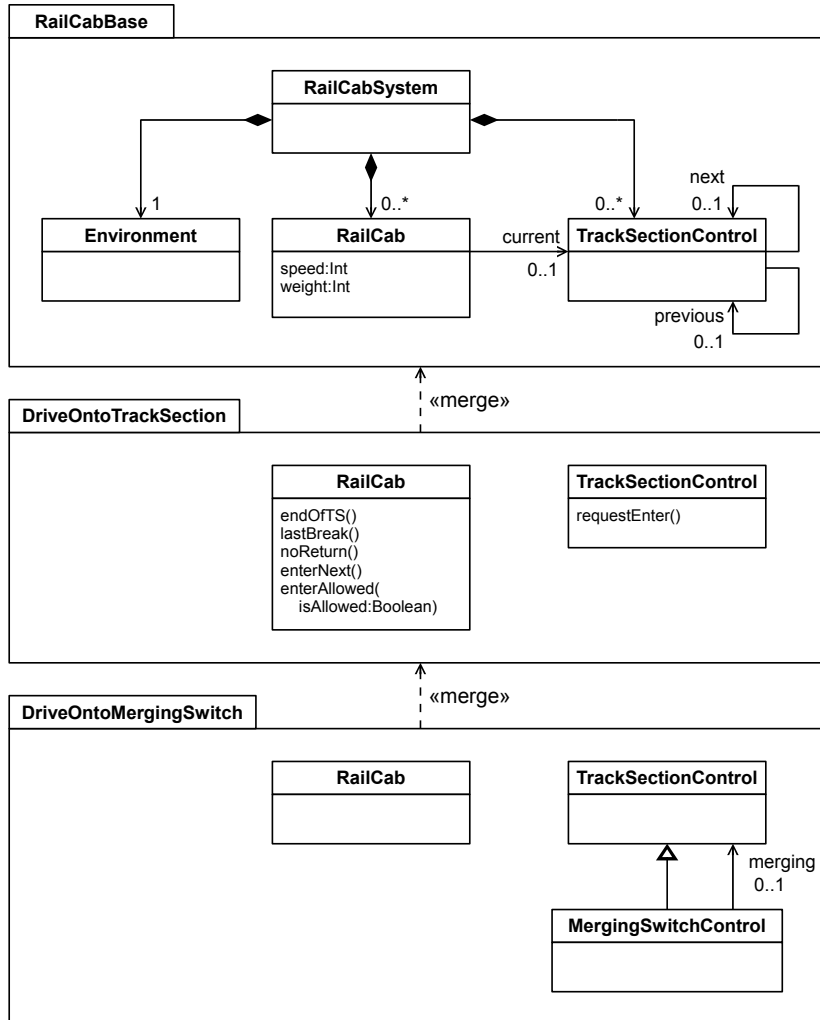


Figure 7.4: The class models of the RailCab specification base package (simplified), and the use cases Drive onto track section and Drive onto merging switch

The package for the use case Drive onto merging switch consists of the elements specified in the package and the contents of the merged packages. More specifically, the semantics of package merge is as follows. A package that merges another package also consists of the contents of the merged package. Classes with the same name are merged to the same classes and, furthermore, properties and operations of the same class are merged when they have the same name, type, and the same parameters (in the case of operations). The merged Drive onto merging switch package would thus look like the package shown in Fig. 7.5.

When further use cases are specified, this may result in a package structure as shown in Fig. 7.6. This figure shows the packages created for further use

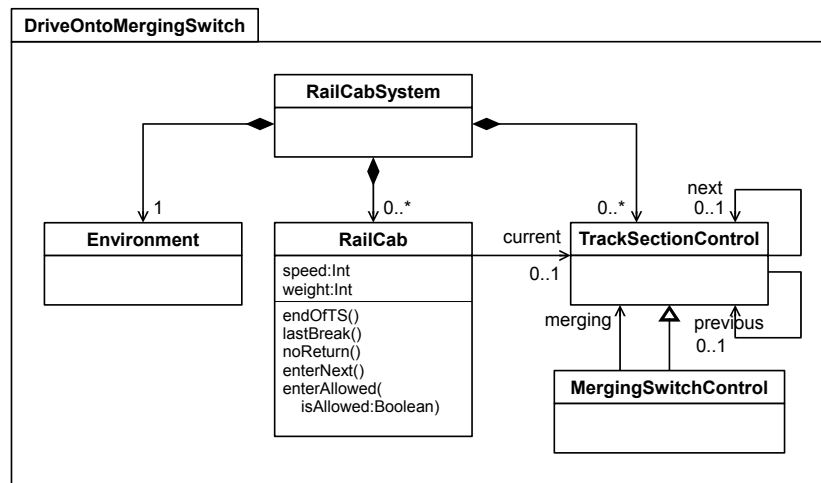


Figure 7.5: The merged package for use case Drive onto merging switch as shown in Fig. 7.4

cases in the RailCab system. The dashed lines represent merge relationships. Here, for example the use case **Hazard occurred** is independent from the use case **Drive onto track section** and therefore only merges the base package. The use case **Enter denied when hazard on next track section**, however, specifies the behavior of the RailCab system where both the use case **Drive onto track section** and **Hazard occurred** occur. That means that this use case will contain MSDs with messages that refer to operations and properties that were specified in the packages of the use cases **Drive onto track section** and **Hazard occurred**. Therefore, the package of the use case **Enter denied when hazard on next track section** merges the packages of the use cases **Drive onto track section** and **Hazard occurred**.

The figure shows further packages of other use cases. These contents of all these packages can later be merged into one package, which is called **RailCab integrated** here. This package, as well as the base package, contain no MSDs. (The packages that contain MSDs and collaboration diagrams are annotated by the small ellipse symbol in the upper-right corner of the package symbol in Fig. 7.6.) We also call a package that merges the packages of other use case specifications an *integrated model* in the following. Based on the integrated model, we can then create objects systems, based on which we can then simulate the specified behavior. Alternatively, we could just merge a subset of the packages, for example to create a specification for a variant of the system or to analyze only a subset of the use case specifications. The package **Enter denied when hazard on next merging switch** is such an example. Here we merge only the packages of the use cases that are involved when a RailCab is about to enter a merging switch on which a hazard has occurred.

7.3.2 Simulation

In order to create a simulation from an integrated model, two steps are necessary. First, the merging package must be translated into an ECore model, see Fig. 7.7

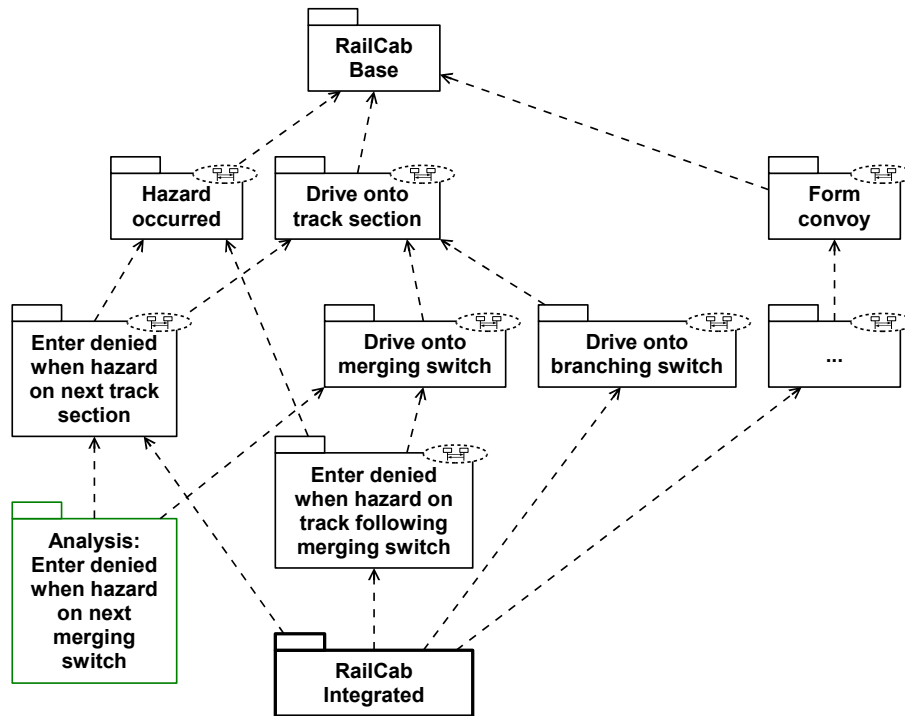


Figure 7.6: The package merge relationships between the packages of some use case specifications in the RailCab system specification

(2). ECore is the meta-modeling language of EMF, which complies to a subset of the Meta Object Facility (MOF) specification of the OMG [MOF06]. This translation is performed automatically via a TGG transformation by the TGG INTERPRETER. In the ECore model, all the classes are effectively merged, which means that there are no merge relationships as in the UML model; the classes that result from these merge relationships (as for example shown in Fig. 7.5) are computed by the transformation. The TGG transformation not only creates the ECore classes, but also stores a correspondence model, which contains the information of which ECore class corresponds to which classes in the UML specification.

Figure 7.7 shows an overview of the SCENARIOTOOLS modeling and simulation process. The top schematically shows a UML specification with packages and their merge relationships (1) and the transformation into a merged ECore class model (2).

Based on the ECore model, a simple editor can be easily created by the Eclipse Modeling Framework (EMF), which allows the engineer to create an instance model of the considered system see Fig. 7.7 (3). Graphical representations of this object system can easily be generated using the Graphical Modeling Framework (GMF)⁷. Remember that there are systems for which many different instances exist. There may be for example many different RailCab track

⁷<http://www.eclipse.org/gmf/>

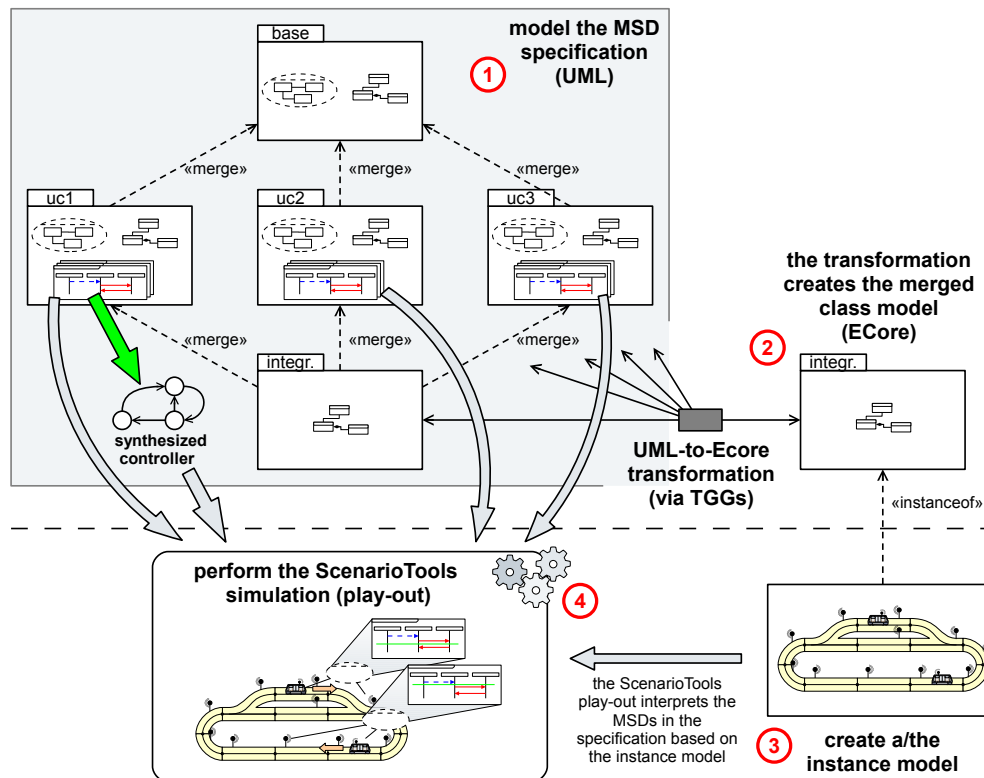


Figure 7.7: An overview of the SCENARIOTOOLS modeling and simulation process

systems with various RailCabs driving in different places. In such cases, the engineers must decide which instance system to simulate.

Based on the instance model, a simulation can be started in SCENARIOTOOLS (4). In order to do so, the engineer has to load the instance model that shall be simulated. Furthermore, the engineer has to select the correspondence model that was created during the UML-to-ECORE transformation. Through the references to the UML specification that are stored in this correspondence model, the SCENARIOTOOLS simulation can determine which lifelines of the MSDs in the (UML-based) specification can be bound to which objects in the (EMF-based) instance model. The simulation then calculates a list of environment events that can occur for the objects in the instance model. The engineer can now start “playing” the role of the environment by selecting one of the possible events from a list in the user interface, see the bottom list shown in Fig. 7.8.

On the occurrence of an environment event, MSDs may become active. The active MSDs and the bindings of their lifelines to the objects in the instance model are shown to the engineer in a tree-view, see the view on the right in Fig. 7.8.

The play-out implemented in SCENARIOTOOLS supports the dynamic binding of lifelines and OCL binding expressions (see Sect. 3.1.15). The simulation can now simulate the reaction of the system in three different modes that the

engineer can choose. First, the simulation engine may execute the super-step in a random manner. That means that one of the active system events is executed as long as there are active system events left to be executed. Alternatively, the engineer can choose a step-mode where he chooses the active system event to execute next. These events can be chosen from the same view that also displays the environment events, see the bottom list shown in Fig. 7.8.

Furthermore, the engineer can choose to execute the system reaction according to a controller that was successfully synthesized from a use case specification. This is indicated at the left of Fig. 7.7. The play-out then works as described in Sect. 5.2; the concepts presented in Sect. 5.3 are not yet implemented.

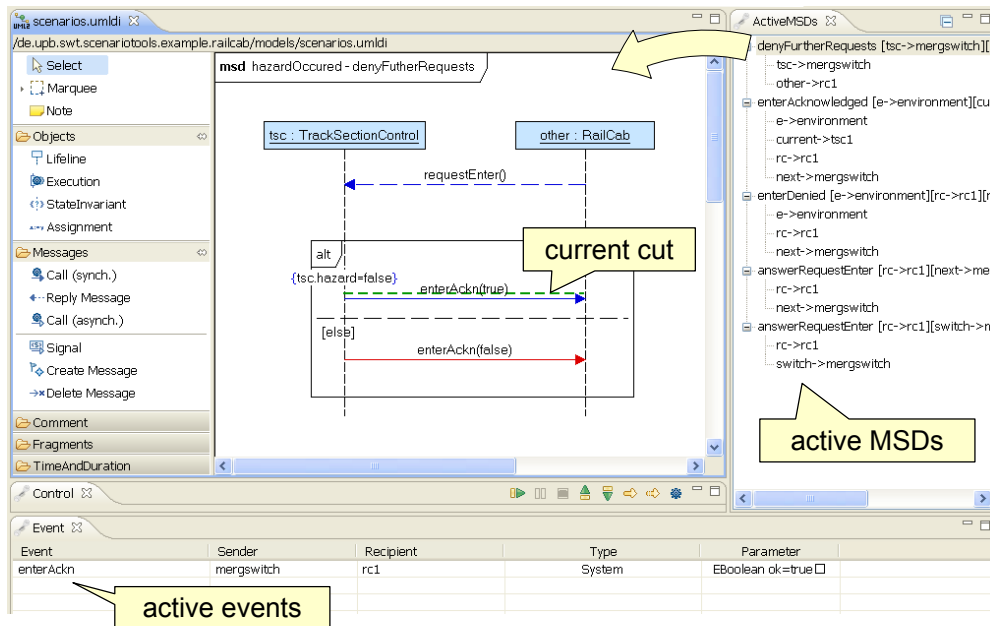


Figure 7.8: An overview of the MSD editor in SCENARIOTOOLS and the simulation user interface.

Similar to the PLAY ENGINE by Harel and Marely [HM03], the simulation in SCENARIOTOOLS can also visualize the current cut of the active MSD in the MSD editor. The cut is shown as a green line, see the middle of Fig. 7.8. The history of the simulation can also be shown in a list view.

The SCENARIOTOOLS play-out algorithm currently only supports the play-out of untimed MSD specifications. However, it supports parameterized messages, alternative fragments, the assignment and evaluation of instance properties, and, as mentioned above, it supports the evaluation of binding expressions given in the form of OCL. It turned out that extending the simulation with the notion of real-time is not a trivial task, and could unfortunately not be realized in the scope of this thesis.

Controller strategies for untimed MSD specifications that were synthesized with UPPAAL TIGA can be integrated with the play-out in SCENARIOTOOLS (see Listing C.1, p. 254 in Sect. C.2.4 for an example of the format in which such

a strategy is generated by UPPAAL TIGA). Because the play-out does not yet support a timed setting and the synthesis does not yet support parameterized messages, this integration is still minimal.

In the future it is also planned to extend the simulation so that the user is kept from selecting environment events that lead to a safety violation in an assumption MSD.

7.3.3 Physics-engine and visualization

The user interface of the SCENARIOTOOLS simulation displays the current state of the simulation in a very generic way. Therefore, depending on the aspects that the engineer is interested in, other ways of visualizing the simulation are desirable. For the visualization of RailCab systems, a 3D visualization was created in the SCENARIOTOOLS VISUALIZATION project, see Fig. 7.9 for a screenshot.

The SCENARIOTOOLS VISUALIZATION project was realized in a cooperative student project in the course “Distributed Software Development”⁸ by master students from the University of Zagreb, Croatia, Mälardalens University, Sweden, and the University of Paderborn.

In this visualization, the movement of RailCabs along track sections and over merging and branching switches is shown in a graphically appealing way. The engineer can now for example observe where a RailCab has to stop because a hazard has occurred on a track section. In the future, this visualization could be extended to show more information about the current state of the simulation in adequate ways, for example the current communication relationships between the RailCabs and track section controls, obstacles on the track sections, or states of the RailCab, such as malfunctions or convoy modes.

In addition to the visualization, a simple physics-engine was implemented within the SCENARIOTOOLS VISUALIZATION project. This physics-engine simulates the movement of the RailCabs along the track section. Now the user does not have to select environment events manually. For this purpose, a simulation of the kinematics of an accelerating and decelerating RailCab was implemented in the physics-engine.

7.4 Evaluation

Within this thesis, powerful techniques were developed and prototypically implemented that support the engineer in the consistent scenario-based design of mechatronic systems. Based on a range of examples, these techniques were evaluated.

First, the *practicality* of the MSD formalism was demonstrated by specifying a comprehensive set of interrelated example use cases from the context of the RailCab project. Second, the *performance* of the synthesis was measured for various timed and untimed example MSD specifications. For these examples, also the performance of the TGG-based transformations was measured.

⁸<http://www.fer.hr/rasip/dsd>

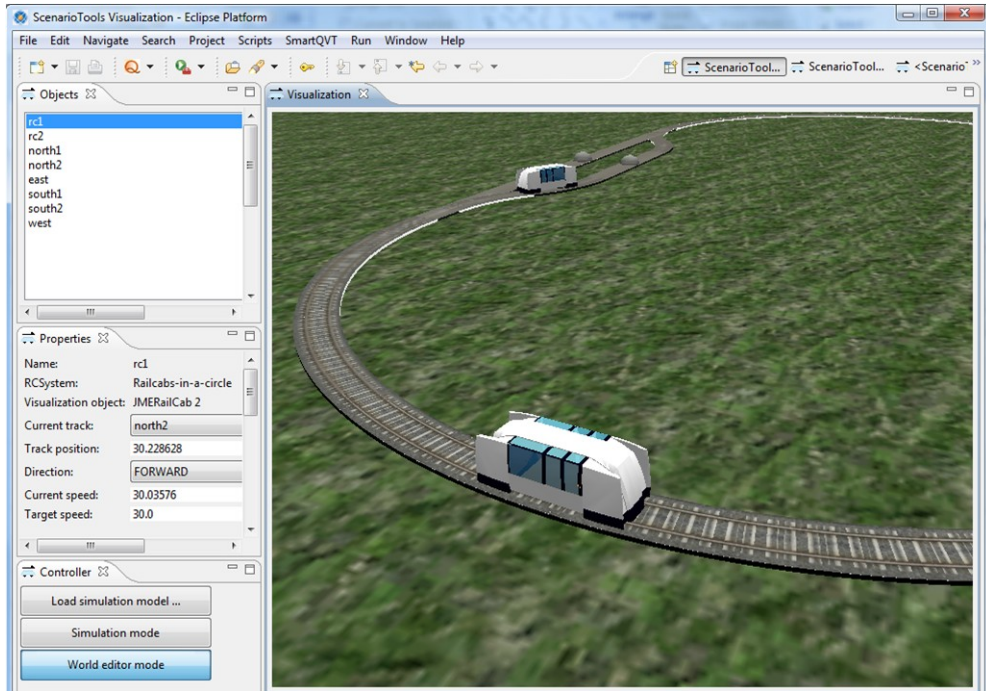


Figure 7.9: A screenshot of the user interface of the 3D visualization that was realized within the SCENARIOTOOLS VISUALIZATION project

A general insight from elaborating these examples is that, although MSDs constitute a convenient formalism, *inconsistencies are easily introduced in scenario-based specifications*, especially in timed MSD specifications. This strongly underlines that synthesis and simulation techniques are *vital* for detecting such flaws in the specification early. The encountered inconsistencies were very often of such a nature that they could be resolved by formulating reasonable environment assumptions. This moreover emphasizes the importance of being able to formally capture specific scenarios that describe what we assume will or will not happen in the environment.

The evaluation of the practicality of the MSD formalism is discussed in Sect. 7.4.1. This section also explains why which kinds of inconsistencies are easily introduced in scenario-based specifications, why environment assumptions are important, and why synthesis and simulation techniques are vital. The results of evaluating the synthesis and simulation technique are discussed in Sect. 7.4.2 and 7.4.3. Last, results of evaluating the TGG transformation technique are presented in Sect. 7.4.4.

7.4.1 Practicality of the MSD formalism

Within the scope of this thesis, a number of timed and untimed example MSD specifications were elaborated:

- The small timed and untimed RailCab use case specifications that were presented in Chap. 4.
- The comprehensive untimed specification of the RailCab presented in Sect. C.1 and C.2
- The timed specification of the production cell presented in Sect. C.3
- The benchmark example specifications presented in Sect. C.4

One general result from elaborating these examples is that MSDs seem to be a convenient formalism for specifying complex interactions. The RailCab example presented in Sect. C.1 in particular demonstrates that it was possible to specify multiple interdependent use cases, which contributed requirements for different situations during the operation of the RailCab system. These use case specifications could then be successfully composed to a complex specification of the RailCab system behavior.

Especially the concepts of *symbolic messages* and *message unification* (see Sect. 3.1.10) were very useful to first specify more abstract scenarios that could later be refined by more specific ones. For example, the MSD `RequestEnterAtEndOfTrackSection` in Fig. C.4 specifies that, after the RailCab requests the track section control for the permission to enter, the track section control must send a positive or negative reply (`enterAllowed(true/false)`). The concrete decision yet remains undetermined in this scenario, in anticipation of various special cases that could require either a positive or negative decision. Later on, MSDs were added to the specification that determined the decision in specific scenarios. Consider for example the MSD `EnterDeniedWhenHazardOnBranchingTrackSection` in Fig. C.11. This scenario describes a case where the next track section control is a branching switch control and where there is a hazard on the track section that is connected to the branching exit of the switch. In this case, the RailCab must not be allowed to enter the track section, which is modeled by the message `enterAllowed(false)`.

In these cases the *hot/cold modalities* of the MSDs were especially helpful to describe possible default behavior, i.e., something that the system can do if there is no reason against it, and certain strictly required behavior that must happen in a specific scenario. For instance, the MSD `DefaultEnterAllowed` (see Fig. C.6) uses a cold message to specify that, by default, the track section control should reply `enterAllowed(true)` in the above example. By contrast, the above-mentioned MSD `EnterDeniedWhenHazardOnBranchingTrackSection` (Fig. C.11) uses a hot message to specify that, in that particular case, the reply must be `enterAllowed(false)`.

However, even though MSDs help the engineer to specify use cases separately from each other, thereby, in principle, being able to concentrate on the specification of one particular aspect of the system's behavior at a time, inconsistencies are easily introduced. This requires a constant effort to detect and resolve these inconsistencies. Many inconsistencies can, however, be avoided if engineers plan and coordinate their specification activities and if extra care is taken with certain MSD constructs, as explained in the following.

Systematically planning and coordinating the specification activities

First, if engineers are going to specify interdependent use cases in separate teams, they are well advised to *coordinate their activities*. If they specify interdependent use case cases unaware of each other, it is likely that inconsistencies or simply naming clashes will occur. It may be that the same classes or events are called differently, which later on will require a tedious renaming in the use case specifications. Moreover, inconsistencies are likely to occur if engineers that concentrate on the specification of one use case are unaware of possible exceptions or alternatives to that case. Then it is likely to happen that sequences of events are modeled as mandatory events (by hot messages), which will later contradict the use cases that specify the alternative sequences of events.

Therefore, engineers will have to *systematically plan* their use case specification activities. The engineers could for example start out with collecting all the use cases, including a very brief informal description, and then elaborate possible dependencies or “overlappings” between them. Then it will often be possible to identify more general use cases and use cases that describe special cases of that general use case, for example as in the scenario above where the RailCab must not be allowed to enter the next track section in certain cases. From such a map of use cases, the engineers could plan their activities and specify the more general use cases first, and more special use cases later. The engineers could for example elaborate a package diagram as shown in Fig. C.2, first adding the more general use cases “at the top”, then the more special use cases “below”, and coordinate their specification activities along the lines in this package diagram.

In addition to systematically planning their specification activities, the engineers should *frequently analyze* their use case specifications in order to detect naming clashes and behavioral inconsistencies among the use cases early.

MSD modeling constructs that should be handled with care

In the process of elaborating these examples, it turned out that inconsistencies are particularly likely to occur if the following constructs are used within MSD specifications.

The system is particularly “vulnerable” to violations of its requirements if requirement MSDs specify *minimal delays*, i.e., hot conditions with a lower time bound. Such a minimal delay means that the system may have to wait for time to elapse before it is able to progress the cut of an active requirement MSD. While waiting, the system listens for environment events, and the cut of the active MSD will be hot. In this situation, the environment has the opportunity to do many things that may violate the active MSD.

Similarly dangerous is the use of *hot environment messages within requirement MSDs*. We use these messages if we wish to express that a certain sequence of events, including environment events, must happen and must not be aborted in any case. The MSD `ArmATransportBlankToPress` in the production cell specification (see Fig. C.26) is an example where a hot environment message is used. If the cut reaches such a message, the system will have to wait for the accord-

ing environment event to occur, and it resides in a hot cut until this message does occur. This again means that the environment has a good opportunity to violate the requirements.

Assumption MSDs are helpful to formulate specific environment assumptions

In these cases, it often turns out that the environment is able to violate the requirements in ways yet unforeseen by the engineer. Sometimes, this will force the engineer to change the system requirements. Very often, however, the engineer will be able to justify that the sequence of environment events that caused the violation of the requirements can be safely excluded from the possible environment behavior. In this case, assumption MSDs are particularly convenient to precisely specify specific sequences of events that are assumed not to occur in the environment. The assumption MSDs *ArmAMoveFromPressToTableTimeAssumption* and *ArmAMoveFromTableToPressTimeAssumption* (see Fig. C.27) for example describe a reasonable assumption on the behavior of the production cell's robot arm that helps to avoid a violation of the MSD *ArmATransportBlankToPress*.

7.4.2 Synthesis

The correctness of the synthesis and its performance were tested using a number of different timed and untimed MSD specifications. The synthesis was applied to the untimed and timed *RailCab* example specifications of Chap. 4, the timed specification of the production cell presented in Sect. C.3, and to the benchmark example specifications presented in Sect. C.4. The main results are the following.

The synthesis technique produces correct results

The correctness of the synthesis technique was not shown formally. However, the synthesis was tested with examples that were diverse in size and nature. For all these examples, the synthesis produced the expected results, i.e., it reported specifications that were expected to be consistent as consistent, and it reported specifications that were expected to be inconsistent as inconsistent.

The synthesis suffers greatly from many interrelated time constraints in timed MSD specifications

Synthesis is an inherently complex problem and its performance especially seems to suffer from many interrelated time constraints in timed MSD specifications. The benchmark example presented in Sect. C.4.3 contained many interrelated time constraints and the synthesis, using the AGAF winning condition, was barely possible with a specification containing six requirement MSDs and one assumption MSD. Note however, that this example is an extreme case, considering the state space and number of time intervals this specification describes compared to the size of the specification, i.e., the number of MSDs, lifelines and messages. The production cell specification is a practical example that also

contains many interrelated time constraints. This example is not yet very big, but already marks the border in the size of timed MSD specifications for which the synthesis, using the AGAF winning condition and no other optimizations (see Fig. C.28), can be carried out successfully.

The synthesis performance is increased in a setting where the system is not allowed to delay the execution of active events

The performance of the synthesis can be immensely increased if we introduce a restriction for the system, demanding that it must not delay the sending of active messages. In the example of the production cell, this restriction could greatly reduce the synthesis times, in some cases the synthesis was by up to a factor of 1500 (using the AG winning condition) faster than without that restriction (see the table in Fig. C.29 compared to the table in Fig. C.28).

Unfortunately, this restriction will make it impossible for the synthesis to find a winning strategy if, in order to satisfy the requirements, the system must delay the sending of an active message. So this restriction does not allow us to find winning strategies for the benchmark example in Sect. C.4.3 or the timed RailCab use case specification presented in Sect. 4.5. (See also the discussion on this restriction in Sect. 4.7.1).

Decomposing the synthesis problem can greatly reduce the complexity of the synthesis problem

Decomposing the synthesis problem as described in Sect. 4.6 can greatly reduce its complexity. This technique was successfully applied to the specification of the production cell and the sum of the synthesis times for the two part specifications compared to the synthesis times for the global specification could be reduced by an average factor of 350 (see the tables in Fig. C.28 and C.32).

Note that how much the total synthesis times can be reduced depends on how “good” a decomposition of the specification can be found. Also note that, due the restrictions of the compositional synthesis technique, it will not be able to apply the technique to all kinds of specifications, and finding a “good” decomposition of a specification is a creative, manual process that also takes time (see Sect. 4.6).

7.4.3 Simulation

The implementation of the play-out algorithm in SCENARIOTOOLS can successfully execute the MSD specification presented in Sect. C.1, and helps in detecting inconsistencies in the specification. The concepts for combining the play-out of a specification with controllers synthesized from single use cases in this specification was validated by the example use case specification presented in Sect. C.2. In combination with the controller synthesized from this use case specification, the simulation can successfully avoid the avoidable violations in occurrences of that use case.

Due to the current limitation of the synthesis and the SCENARIOTOOLS play-out algorithm, a more comprehensive evaluation of the symbiosis of synthesis

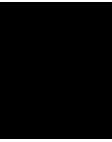
and simulation was not yet possible and is subject of future work. No detailed performance measurements were conducted for the implementation of the play-out algorithm, since its performance was not the focus of this thesis.

One result from simulating the example MSD specifications is that the usability of `SCENARIOTOOLS` yet suffers from an insufficient visualization of the simulation. Even though a 3D simulation can be used to simulate the Rail-Cabs' movement in a track system (see Fig. 7.9), and the cuts of active MSDs can be visualized (see Fig. 7.8), it is yet difficult for the user to maintain an overview of complex interactions that take place. This could be improved by a better graphical representation of the message exchange that takes place and the possibility to replay certain steps in the simulation, i.e., to step back and then forward again in the simulation.

7.4.4 TGG-based model transformation

The complex MSD-to-TGA mapping could be successfully specified and executed based on TGGs (see Appendix B). The performance measurements show that the `TGG INTERPRETER` is not very fast, but exhibits acceptable transformation times for the considered examples. For example, the largest MSD specification that was transformed into an MSD system was the decomposed production cell specification, consisting of two packages, 15 MSDs, 35 lifelines, 47 messages and 13 time conditions. It took about 37 seconds to transform this specification into the two corresponding TGA systems (see the table in Fig. C.32). The transformation time grows linearly with the size of the MSD specification, as was shown by the help of the benchmark examples (see Sect. C.35).

Overall, the performance of the `TGG INTERPRETER` is acceptable. Keep in mind, first, that applying TGG rules involves expensive pattern matching, second, that TGGs are a declarative formalism and the `TGG INTERPRETER` will often try to apply rules that, after matching parts of the rule's graph pattern, turn out not to be applicable. Third, the `TGG INTERPRETER` relies on interpreting the TGG rules, so no code is generated from the TGG rules as in `FUJABA` or `MOFLON`, which could potentially be executed faster. Nevertheless, the performance of the `TGG INTERPRETER` could probably yet be increased by applying smarter heuristics, which avoid many of the unsuccessful rule application attempts. Improving the performance of the `TGG INTERPRETER` is yet future work.



Related Work

This section discusses the related work of this thesis. An overview of existing scenario-based design approaches was already given in Sect. 2.4. This section therefore concentrates on work that is more directly related to the concepts presented in this thesis. Section 8.1 presents advanced related play-out techniques, and Sect. 8.2 discusses related synthesis approaches. Especially, Sect. 8.2.2 discusses existing approaches for synthesizing distributed implementations from LSC specifications in order to highlight possible directions for future work. Advanced TGG concepts that are related to the TGG extensions presented in this thesis were already thoroughly discussed in Chap. 6. Therefore, this discussion is not continued here.

8.1 Advanced play-out techniques

Smart play-out [HKMP02b, HKMP02a, HKP04] is an extension of the play-out algorithm, implemented for an improved play-out of LSCs in the PLAY ENGINE [HM03]. Smart play-out allows the play-out algorithm to avoid some avoidable violations by looking ahead a limited number of steps that the system takes in reaction to an environment event. For this purpose, smart play-out relies on model-checking. Each time that the system must react to an environment event, the played-out LSC specification and the current state of the system, i.e., object properties and state of (pre-)active LSC copies, is encoded into a model-checking problem. Model-checking is then used during execution-time to calculate a sequence of steps that the system can take in reaction to an environment event until there are no active messages left (i.e., enabled messages in the main chart of active LSC copies).

Smart play-out can only avoid a limited number of avoidable violations, because it can only look ahead in the scope of one *super-step*, which is the sequence of steps the play-out algorithm takes in reaction to an environment event, until it waits for the next environment event to occur. In a timed setting, super-steps can also end where the system waits for minimal delays to become

true; therefore, in specifications with many minimal delays, the super-steps may thus become very short, so that the power of smart play-out is especially limited. Especially in a timed setting, the symbiosis of synthesized controllers with the play-out algorithm therefore seems an especially promising approach in more drastically reducing the number of avoidable violations during play-out. Unfortunately, these concepts were not yet elaborated in this thesis to such an extent that this assumption could be validated.

Harel and Segall propose another approach for smart play-out that does not rely on a model checker, but instead on a planning algorithm [HS07]. This approach, however, has the same limitation in the number of steps it can look ahead. Therefore, the approach of combining play-out with synthesized controllers for parts of the specification, as presented in Chap. 5, will be able to avoid violations that smart play-out will not be able to avoid. It would be interesting to investigate in the future if the two approaches, smart play-out and the symbiosis of simulation and synthesis, can be beneficially combined. For example, if a number of single use case specifications, which were each shown to be consistent by synthesis, overlap during the simulation, avoidable violations may still occur, even if the simulation is guided by the controllers synthesized for the single use case specifications. Here maybe the concept of smart play-out can help of avoiding some of these avoidable violations.

One particular drawback of smart play-out is that it requires a static object system in order to encode the problem of finding an admissible super-step in a model checker. The approach presented in this thesis, combining play-out with synthesis, does not have this limitation.

The symbiosis of play-out and synthesized controllers from (parts of) the specification has not been elaborated thus far. Kugler et al. [KPP09] have integrated their synthesis approach with the PLAY ENGINE to execute the resulting implementation. Their work, however, does not consider how a synthesized controller can be executed *in combination* with not yet synthesized parts of the specification. Also is it not considered how multiple controllers, synthesized for subsets of the specification, can be executed in combination.

Maoz and Harel present an alternative realization of the play-out algorithm by mapping LSCs to executable AspectJ code [MH06]. The implementation contains extension points for strategies to be included that advise the play-out algorithm on which steps to choose if there is a set of active events. Thus far, however, no such extensions have been elaborated.

8.2 Related synthesis approaches

There exists a range of synthesis approaches that propose techniques to synthesize a global controller from universal LSC specifications and, thereby, to check the consistency of the specification. They are closely related to the synthesis technique in Chap. 4 and will be discussed in the following. It follows a short discussion on approaches that propose techniques to derive a distributed implementation from the specification (Sect. 8.2.2).

8.2.1 Synthesis of global controllers

This section introduces the existing approaches for synthesizing global controllers from LSC specifications by Harel and Kugler [HKMP02b], Bontemps et al. [BS03, BSL04, BH05], Kugler et al. [KPP09, KS09] and Larsen et al. [LLNP09, LLNP10].

Synthesis by product automata construction by Harel and Kugler

Harel and Kugler describe the first approach for synthesizing LSC specifications [HK99, HK02]. In their work, they first present a mapping from simple universal LSCs to finite state machines. Second they present an algorithm that first constructs a product automaton from the finite state machines and then successively eliminates the states from where the system cannot guarantee to always avoid hot violations. If the resulting automaton is empty, the specification is inconsistent.

Third, Harel and Kugler present a mechanism by which a resulting automaton, if it is not empty, can be distributed such that it results a state-based implementation of the objects that satisfies the specification. The distribution approach essentially consists of copying the product automaton for each object in the system and then synchronizing all these automata whenever a step is taken by the system. This results in large state machines for the objects and an extensive overhead of communication among the objects. In principle, however, Harel and Kugler could demonstrate that an LSC specification is realizable by an implementation of the objects iff it is consistent.

Harel and Kugler only support a limited variant of LSCs in their synthesis approach. The synthesis is performed with untimed (universal) LSCs that may only have one message in the prechart, environment messages may only appear in the prechart, and there is no support for assignments, conditions, or parameterized messages. Also a static object system and an iterative semantics (see Sect. 3.1.6) is assumed.

First game-theoretic approach by Bontemps et al.

Bontemps et al. describe the first game-theoretic approach for synthesizing a global controller from an LSC specification [BS03, BSL04]. They regard a restricted subset of universal LSCs in which there are no conditions or other rich constructs. Also the messages have no distinct modality—all messages appearing in the main chart must occur, which corresponds to the meaning of hot executed messages in this thesis. In addition to universal LSCs, a specification may contain an *initial* LSC, which contain sequences of events that must occur at the beginning of each execution of the system.

From the specification, a Büchi automaton is created that accepts all infinite sequences of steps that respect the safety and liveness properties of the universal LSCs. An algorithm then considers a turn-based game on this automaton and tries to synthesize a winning strategy, i.e. a strategy for the system to always take steps in the automaton such that the resulting run is accepting.

The concepts were prototypically implemented [BH05]. Extensions to consider timed specifications were not elaborated.

(Compositional) synthesis by Kugler et al.

Kugler et al. describe a game-theoretic synthesis technique [KPP09, KS09] for synthesizing a global controller from an LSC specification. Their idea is very similar to the concepts proposed by Bontemps et al. They adopt the encoding from LSCs to an SMV (Symbolic Model Verifier) model that was elaborated for smart play-out [HKMP02b]. Based on this model, their algorithm calculates a winning strategy for the system if the specification is inconsistent. If no winning strategy exists, the specification is consistent. In contrast to the concepts presented by Bontemps et al., the approach by Kugler et al. is the first realization of a game-theoretic synthesis approach that uses a symbolic representation of the state space. However, only a limited number of LSC language features is supported. The LSCs are untimed, and there is no support for parameterized messages or object properties. Again, a static object system and an iterative semantics of the charts is assumed.

As already mentioned in Sect. 8.1, Kugler et al. integrate their synthesis in the play-engine in order to execute the global controller. After each environment event, a super-step is extracted from this global controller by using a model checker, just as in smart play-out [HKMP02b].

Kugler and Segall have furthermore proposed a *compositional* approach for synthesizing controllers from LSC specifications [KS09]. However, their compositional approach differs significantly from the compositional approach presented in Sect. 4.6. Kugler and Segall propose the following approach. First, they use the above-mentioned technique [KPP09] to synthesize controllers from arbitrary disjoint subsets of the MSDs in a specification. If, in this process, it turns out that there is an inconsistency among one of these subsets, this implies that the complete specification is inconsistent, and therefore this inconsistency must be resolved. If all the subsets of MSDs are consistent, this yields a set of intermediate controllers. From subsets of these controllers, yet other, bigger controllers are then synthesized similarly to the approach presented by Harel and Kugler earlier [HK02]. This process continues until a global controller is synthesized. The benefit of this approach is, first, that inconsistencies may be found quickly already during the first runs of the synthesis. Second, the complexity of the synthesis task is reduced slightly, compared to running the synthesis for the whole specification at once. Intuitively, each synthesis steps eliminates more and more inadmissible runs on each level. Therefore, during the last synthesis steps, it may be that the state space that must be considered is smaller than the state space that would need to be considered when synthesizing a controller from the complete specification at once.

By contrast, the compositional approach discussed in Sect. 4.6 requires an engineer to decompose the specification in a smart way, using an assume-guarantee paradigm. If then controllers can be synthesized successfully for the parts, this implies that the original specification is consistent, and the combination of the two controllers form an admissible implementation of the speci-

cation. This approach requires more creativity from the engineers, but it will in many cases be possible to reduce the complexity of the synthesis problem more drastically than by the approach of Kugler and Segall. Especially, this is because the compositional synthesis approach described here does not require further synthesis steps on the controllers that were synthesized from the part specifications.

Synthesis via Uppaal Tiga by Larsen et al.

Larsen et al. presented an approach for synthesizing controllers from timed universal LSC specifications via UPPAAL TIGA [LLNP09, LLNP10]. The approach was developed in parallel with the synthesis technique presented in this thesis [Gre09, Gre10]. There are some similarities in both approaches, but also many differences.

Similar is that Larsen et al. synthesize controllers from timed or untimed universal LSC specifications. The LSCs have synchronous messages and the timed variant may have clock resets and time conditions, similar to the timed MSDs in this thesis. The LSCs are mapped to a network of Timed Game Automata (TGA). The winning condition used by Larsen et al. is a simple safety condition to always avoid hot violations, similar to the AG condition presented in Sect. 4.4.2.

Their mapping principle is as follows. First, each lifeline of each universal LSC is mapped to an automaton, called *lifeline automaton*. Each location on the lifeline is translated into a location in such an automaton. Between these locations are edges that send and receive over particular broadcast channels that represent according events that are enabled when the cut of the MSD is at the corresponding location. Additionally, there is an automaton created for each universal LSC, called *coordinator automaton*, which for example synchronizes the lifeline automata when all the lifelines synchronously progress from the prechart into the main chart.

The sending of a message is represented by taking an edge in a lifeline automaton which emits over a broadcast channel that represents this event. All other lifeline automata where this event is “enabled” have an edge which receives over this channel, and thus, these lifeline automata then progress synchronously.

The messages sent by system instances are represented by controllable edges. Therefore, the system may “produce” such events that are currently enabled. Similarly, the environment messages are represented by uncontrollable edges in the lifeline automata corresponding to environment instances. This encoding has a severe limitation: the environment may *only produce events that are currently enabled in an LSC*. That means that environment events can only occur if there is currently an active LSC that “expects” this environment event to occur; environment events that are not enabled in any LSC will not occur. The latter assumption is overly optimistic, and has no practical justification.

In the approach presented in this thesis, by contrast, any environment event may always occur, unless it is explicitly restricted by environment assumptions. The approach described in this thesis thus does not accidentally introduce overly

optimistic assumptions on the environment, but allows the engineers to specifically define which behavior will or will not occur in the environment.

In the approach by Larsen et al., liveness in the main chart is expressed by marking the locations in the lifeline automata which represent lifeline locations in the main chart as *urgent*. This means that when the system is in such a location, time is not allowed to pass. This, however, implies that no time is allowed to pass between the sending of enabled events. The system may only delay the evaluation of time conditions. Therefore, this synthesis approach would fail in finding a winning strategy for the example RailCab use case *Drive onto merging switch*, which requires that the sending of the event `enterAllowed` is delayed. The synthesis technique presented in this thesis, by contrast, manages to synthesize an admissible controller for that specification.

One characteristic of the approach by Larsen et al. is that they consider the *invariant* interpretation of the LSCs (see Sect. 3.1.6). The invariant interpretation allows multiple active copies of an LSC to be created for the same instances. This may happen when the triggering event sequence of the main chart occurs repeatedly in the chart. However, the encoding proposed by Larsen et al. does not actually capture these different copies. Rather, occurrences of the first event always only re-initialize the active copy of an LSC.

Finally, by the winning condition used by Larsen et al., controllers may be synthesized where the system remains in an infinite loop. That may for example happen if there are two MSDs where one MSD always triggers the activation of the other. Using the AGAF winning condition in the approach presented here, this can be avoided. (See also the discussion on the different winning conditions in Sect. 4.4.2.)

8.2.2 Synthesis of distributed controllers

The synthesis approaches discussed above are restricted to synthesizing global controllers from the LSC specifications. The same restriction applies to the synthesis technique elaborated in this thesis. In practice, however, a global controller cannot always be used directly as the implementation, especially in distributed systems with multiple controllers. Therefore, it is ultimately desired to have a method for automatically synthesizing a distributed implementation of the specification if the specification is consistent.

However, synthesizing a distributed implementation is a problem that has not been solved in a satisfying way. Bontemps et al. have actually shown that the problem of deciding whether there exists a distributed implementation of an LSC specification is not decidable [BS05]. But, there exist approaches for synthesizing distributed implementations that are either incomplete or make additional assumptions.

The approach by Harel and Kugler mentioned above [HK99, HK02] proposes a technique to construct a distributed implementation by essentially copying the global controller for each object. Additional messages are introduced that synchronize all objects upon each step of the system. However, especially in time-critical applications, where the communication between the components of a system may take time, this approach may not always be applicable.

Another incomplete method is proposed by Bontemps et al. [BS03, BSL04]. They propose a technique where the implementation of a system component is generated from the universal LSCs using a mapping to automata. Each lifeline of each universal LSC is projected onto the system component that it represents. Then an implementation is created from these lifelines such that a component tracks the messages that it receives and then sends messages similar to a play-out algorithm that just views the system from the perspective of only that component. That means for example that a component may decide to send a message that causes a hot violation for some other component. Last, it is checked whether a distributed implementation created this way satisfies the specification, i.e. if violations like the one mentioned above do not occur. If the implementation satisfies the specification, the created system is a valid implementation of the specification. If the created system does not satisfy the specification, there may still be another implementation satisfying the specification. Thus the approach is not complete.

Similarly, Harel et al. propose a synthesis technique that mainly relies on a mapping from universal LSCs to statecharts [HKP05]. For each component in the specification, a statechart is created with parallel regions for each LSC in which the component is represented by a lifeline. Each region contains a sub-statechart which encodes the behavior of the component in an LSC, i.e., the messages it sends and receives. The states in the sub-statecharts represent locations in the chart. The resulting statecharts are synchronized by additional messages when the cut in a chart synchronously progresses from the prechart to the main chart for all components in the LSC. Ultimately, it is checked if the implementation resulting from this mapping satisfies the specification. More specifically, model checking is employed to check whether the resulting implementation can infinitely often reach a state where all charts are simultaneously inactive. Again, the approach is not complete, i.e. the specification may nevertheless be consistent and there may exist an implementation for the specification that is just not discovered by the approach. The approach presented by Harel is limited only to LSCs with simple messages. Lochau [Loc07] proposes extensions to that approach.

There have also been other methods proposed for distributing a global controller to a network of local controllers. One recent approach by Halle and Bultan [HB10] suggests a projection from a global controller to distributed controllers and to later check the equivalence of the two behaviors. The novelty of this approach is that the projection works in such a way that each component also “guesses” what the state of the other components in the system may be. This way, a distributed implementation may be found that behaves exactly like the global controller. The approach is also not complete, i.e., there may exist a distributed implementation if the result of the approach is negative. It is an interesting future research perspective to investigate if such techniques can be applied for distributing the global controller resulting from the synthesis technique presented in this thesis.

Conclusion and Future Research

This thesis presented a comprehensive technique for the scenario-based design of advanced mechatronic systems that allows engineers to precisely specify and compose the scenario-based requirements of a system and to analyze these requirements for inconsistencies already during the early design of the system. The core concepts of this technique were implemented in the SCENARIOTOOLS tool suite and were validated by a range of examples. The technique prevents costly iterations in the system's development and devastating flaws in the final product that may occur if inconsistencies are detected only in later development phases or remain undetected until the system's start of operation.

9.1 Summary

The technique developed in this thesis is based on the language of Modal Sequence Diagrams (MSDs) [HM08], a formal interpretation of UML sequence diagrams based on Live Sequence Charts (LSCs) [DH01], a formalism for precisely specifying how the components in a system should, must, or must not interact in reaction to events in their environment. Furthermore, the technique developed in this thesis is based on the play-out algorithm [HM02a, HM03], a technique for executing universal MSDs/LSCs that allows engineers to simulate the scenario-based specification.

This thesis extended the existing concepts by two complementary techniques. First, a novel technique for *automatically synthesizing controllers from MSD specifications* was developed, which allows engineers to effectively detect inconsistencies in timed and untimed MSD specifications. The synthesis is realized by a mapping from UML-based MSD specifications to networks of Timed Game Automata (TGA). By also supplying an appropriate winning condition, UPPAAL TIGA, a tool for finding winning strategies in timed or untimed two-player games

[BCD⁺06, BCD⁺07a], can then determine the consistency of a given MSD specification. In doing so, UPPAAL TIGA calculates an admissible controller for the MSD specification if it is consistent. The particular benefit of this approach is that it reuses the efficient data structures and the efficient algorithm [CDF⁺05] of a validated tool.

The main novelty in the presented synthesis technique is that controllers can be synthesized from specifications that not only consist of scenarios that describe the required system behavior. Engineers may also explicitly formulate assumptions on the possible environment behavior using *assumption MSDs*. The synthesis can then show whether there exists a system that can fulfill the requirements in the specification provided that the environment adheres to the assumptions. The synthesis of LSC/MSD specifications under consideration of environment assumptions has not been studied previously. This extension is however crucial for practical applications in the domain of mechatronic systems. Very often a system will only be able to satisfy certain requirements if it can be assumed that not arbitrary sequences of environment events can occur that may force the system to violate its requirements. In mechatronic systems, the possible sequences of environment events are often restricted due to mechanical principles, the laws of physics, or because certain properties are known about external mechatronic- or software components.

Based on the assumption MSDs, a *technique for the compositional synthesis* of MSD specifications using the assume/guarantee paradigm was developed in this thesis. In certain cases, it allows for decomposing a synthesis problem into two or more smaller synthesis problems that in sum can be solved much faster than the original synthesis problem. Such a technique has not yet been elaborated for LSC/MSD specifications. A sketch for the proof of the soundness of the compositional synthesis technique was presented and a successful application of the technique was illustrated by the help of a simplified MSD specification of an industrial production robot.

Due to the inherent complexity of the synthesis problem, it will not be possible to apply the synthesis to the often considerably large specifications of a mechatronic system. Also, not all MSD specifications can be decomposed and it is not possible to synthesize the specification of a dynamic system, i.e., systems in which communication relationships between components may change during runtime. Nevertheless, it is suggested to use synthesis to detect inconsistencies in parts of the specification. More specifically, synthesis can be used to detect inconsistencies within the specifications of single use cases if these use cases refer to a static structure within a bigger, possibly dynamic system. Controllers can also be synthesized from *composed use cases*, which are occurrences of certain combinations of use cases occurrences that overlap within a particular structural context.

This way, many inconsistencies can be effectively detected. However, the engineer cannot be sure if there exists a system that adheres to the specification if the use cases overlap in unforeseen ways. For dynamic mechatronic systems, or systems with a complex specification, it is therefore suggested to analyze the complete specification for inconsistencies via the simulation based on the play-out algorithm.

The play-out algorithm was extended in the scope of this thesis, so that now the play-out of the scenarios in a specification can be guided by controllers that were successfully synthesized from use case specifications or composed use case specifications. This extension improves the simulation, because it helps the play-out avoid more avoidable violations of the specification (“false negatives”), which may occur during the play-out of MSD specifications that are typically under-specified during the early design. The advantage is that the simulation is less disrupted by avoidable violations and the engineer has more reason to suspect actual inconsistencies when violations occur during the simulation.

To achieve this symbiosis of simulation and synthesis, this thesis presented an extension to the play-out algorithm. In this extension, the execution of the active MSDs that make up an occurrence of a use case are controlled according to a controller that was synthesized from the according use case specification. A prototype validating this extension of the play-out algorithm was implemented in SCENARIOTOOLS. Concepts were furthermore elaborated for controlling the active MSDs that belong to the occurrences of composed use cases. The challenge for the extended play-out algorithm is that it may have to anticipate multiple different possibilities of how the situation described in the composed use case specification may emerge from occurrences of its constituent use cases. This poses a complex task for the play-out algorithm that may become infeasible if controllers for many complex composed use cases shall guide the play-out.

Finally, in the process of developing the above-mentioned synthesis technique, new extensions to TGG-based model transformations were developed and implemented within TGG INTERPRETER. The mapping from UML-based MSD specifications to a TGA system in UPPAAL TIGA posed a major model transformation challenge. Complex structures in stereotyped UML sequence diagrams had to be translated into a complex TGA model, consisting of function and variable declarations as well as automata definitions. For this purpose, the Object Constraint Language (OCL) [OCL10] was integrated with TGGs for expressing application conditions and attribute value constraints. In UML models, furthermore, constraints can now be formulated over stereotype applications on UML elements and the values of the additional attributes that a stereotype application supplies to the UML element. Furthermore, a *rule generalization* (inheritance) mechanism was introduced, improving the concepts originally proposed by Klar et al. [KKS07]. These extensions made it possible to formalize and implement the complex MSD-to-TGA mapping in a concise way by a visual and rule-based formalism.

9.2 Future research

This thesis presented an ambitious vision for an improved scenario-based design of mechatronic systems. Many novel and useful concepts were thoroughly elaborated, prototypically implemented, and validated by examples. In order to solve a broader range of interesting practical problems, however, there are some open tasks.

The synthesis, for example, will have to be extended to synthesize controllers for MSD specifications that contain MSDs with parameterized and asynchronous messages as well as richer constructs like if-then-else fragments or while loops. Also, for certain kinds of environment assumptions, it will be relevant to consider during the synthesis that the system may not have the complete information about the state of the environment, but can only observe part of the events occurring in the environment.

Furthermore, the play-out algorithm will have to be extended to simulate real-timed MSD specifications. Also the concepts for guiding the play-out by controllers synthesized from composed use cases must yet be implemented and validated. In order to support the engineer in better understanding the inconsistencies that may be detected during the synthesis of use case specifications, it would also be desirable to extend the SCENARIOTOOLS simulation such that it can execute the counter-strategy that will be synthesized by UPPAAL TIGA in that case. Using the SCENARIOTOOLS simulation to “play” the role of the system against an environment that will be always able to violate the requirements, the engineers may be able to better understand the nature of the inconsistency.

It is also desirable to integrate SCENARIOTOOLS with the MECHATRONIC MODELER [GBRP10, GDN10], a tool for the design of mechatronic systems based on the interdisciplinary specification language that is developed within the scope of the CRC 614.

In the future it is particularly interesting to investigate how to further reuse the formal scenario specification when the software of the system must be implemented in later development phases. One idea is to use the scenarios to generate test cases that support the software engineers in the manual implementation of the software.

In some cases, if the mechatronic system is controlled by a single processor, the software for the system could even be automatically generated from a global controller if one can be successfully synthesized from an MSD specification by the synthesis technique presented here. In a mechatronic system with distributed processors, however, that will not be possible. For these cases it should be investigated how to automatically synthesize distributed controllers from the specification.

Some ideas have been presented previously (see also Sect. 8.2.2). For example, Harel et al. describe how a global controller that could be successfully synthesized from a specification can be mapped to local controllers for objects in an object system. This approach, however, introduces an immense communication overhead, because additional messages for communicating the global state among the local objects are introduced [HK02].

Another interesting approach for distributing a global controller was presented by Halle and Bultan [HB10]. They describe how to project a global controller to local controllers for system objects that “guess” the global state (i.e., the state the system would be in according to the global controller) from the messages they send and receive. If these guesses of the global states can deviate from the actual global state, the local controllers do not constitute a valid implementation of the global controller. But there could be ways to successively add coordination messages that eventually allow the local controllers to always

correctly agree on the global state of the system, thus finally constituting a valid implementation. Such an approach could be used to create a distributed implementation for objects in a system from a global controller that could be synthesized from an MSD specification using the technique presented here.

One open question regarding the simulation of MSD specifications (see the process described in Sect. 5.1, Fig. 5.1) is if there is any indication of when the engineer has simulated “enough” to be sure, at least to a measurable degree, that the specification is free of inconsistencies. To answer this question, it would be necessary to determine, first, how many of the possible combinations of overlappings of use case occurrences were analyzed and, second, how many of the possible runs in the different combinations of overlappings of use case occurrences were simulated.

Especially in dynamic systems, finding all possible overlappings of use case occurrences can be difficult. It could, however, be possible to capture the possible reconfigurations in a dynamic system in a graph transformation system and enrich this system by the information of when and where which use cases may occur. Then the different possible states of that system can be systematically explored in order to find the different possible overlappings of use case occurrences.

Because some functions in mechatronic systems are of a continuous nature, it would furthermore be desirable to model *hybrid* scenarios, i.e., scenarios that describe not only which sequences of events may, must, or must not occur, but also describe constraints on continuous variables. Similar ideas were already presented in the past [GKS00]. It would be especially interesting to investigate if there are certain kinds of such constraints by which play-out algorithm, i.e., the operational semantics of the scenarios, could be extended so that hybrid scenarios can be executed. Once this is possible, the simulation can be employed to find inconsistencies also in hybrid specifications.

Meta-Models and Profiles

The mapping from an MSD specification to a network of Timed Game Automata (TGA) as explained in Chap. 4 is chiefly realized by a TGG-based model transformation from a stereotyped UML model to an ECore model of a TGA network (see Sect. 7.2, Fig. 7.3). In order to better understand the TGG rule set presented in Chap. B, this chapter first explains the meta-model for networks of TGA in UPPAAL TIGA in Sect. A.1 and, second, presents the profile used for modeling MSD specifications with UML in Sect. A.2.

A.1 The meta-model for Uppaal TIGA

The UPPAAL TIGA ECore meta-model captures the parts of the UPPAAL TIGA language that are used in the synthesis approach presented in this thesis. Instances of this meta-model, i.e., models of concrete networks of TGA, can be translated to a valid input file for UPPAAL TIGA by a model-to-text transformation (as described in Sect. 7.2). A brief explanation of the meta-model is given in the following; see the tutorial on UPPAAL [BDL04] or the online help of the UPPAAL TIGA tool for a more comprehensive description of the language.

The first class diagram in Fig. A.1 shows the meta-classes that are used for describing the automaton templates, the global resp. template-specific declarations, and the declarations of TGA systems (i.e., the instantiation of automaton templates). The root of each model in UPPAAL TIGA is called NTA (network of Timed Automata). It contains a number of automaton templates and a system declaration that defines a number of instantiations of these templates. Each template consists of locations and edges that have one source and one target location. The NTA and each template may have declarations that consist of declarations of functions, channels, clocks, and Boolean and integer variables. Functions may themselves have parameter variable declarations. In this model,

to declare for example a Boolean variable, a `BooleanDeclaration` has to be created together with a `VariableID` that determines the name and, optionally, an initial value for that variable. When multiple variable IDs are contained in a Boolean declaration, this leads to declarations like `bool a = true, b = false;`

Note that not each possible instance of the meta-model leads to valid input for UPPAAL TIGA. For example, according to the meta-model, a channel may be initialized with a String expression, or the parameter variable declarations for functions may have an initializer, which is not possible in UPPAAL TIGA. The aim of this model is primarily to support the MSD-to-TGA mapping in a convenient way. The TGG transformation uses this model in such a way that the XPAND transformation produces valid input files.

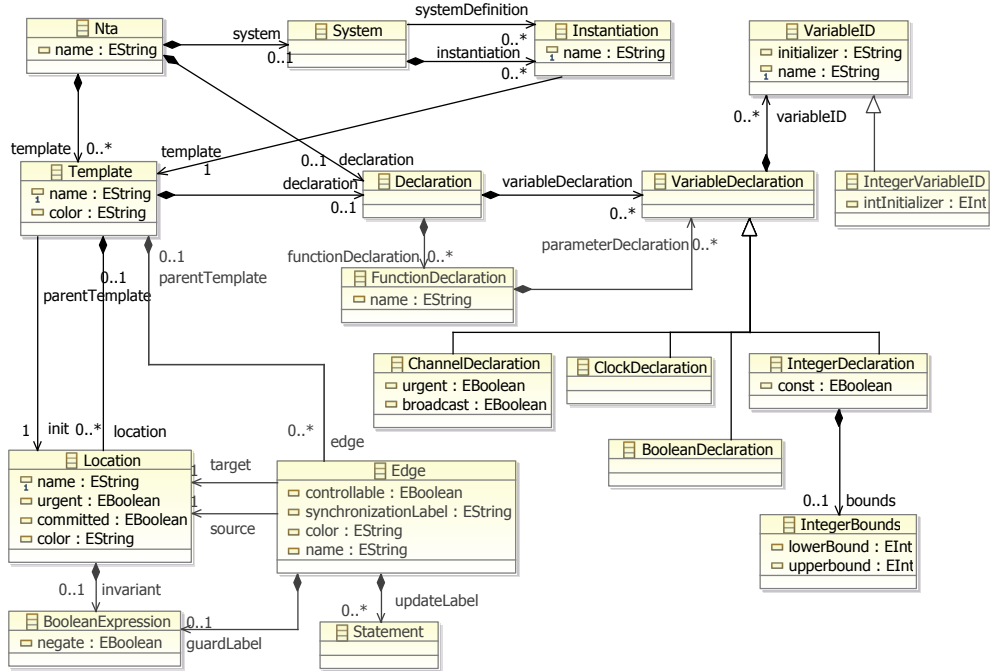


Figure A.1: An ECore meta-model for UPPAAL TIGA (automaton templates, global/template declarations, system declarations)

Most properties of the locations and edges are encoded by String or Boolean attributes. For the invariant expressions of locations and the guard expressions of edges, the model defines the class `BooleanExpression`. Further, the update expression of an edge is represented by a `Statement`. Figure A.2 shows the meta-classes for functions, statements, and expressions. An expression can either be a plain text expression, an integer literal, or a Boolean expression. Again, this is not intended to represent a valid grammar for UPPAAL TIGA expressions, but shall just serve the MSD-to-TGA mapping in a convenient way. A Boolean expression can either simply be an expression given in the form of a text string,

but it can also be a conjunction or disjunction of other Boolean expressions, or a binary compare expression between two other expressions. The compare expressions can use a number of compare operators given in the enumeration `CompareOperator`. A Boolean expression can further be the call to a Boolean function with zero or more arguments. A function in this model may have a number of statements. In the scope of this paper, however, only Boolean functions are created with a single return statement that is a Boolean expression.

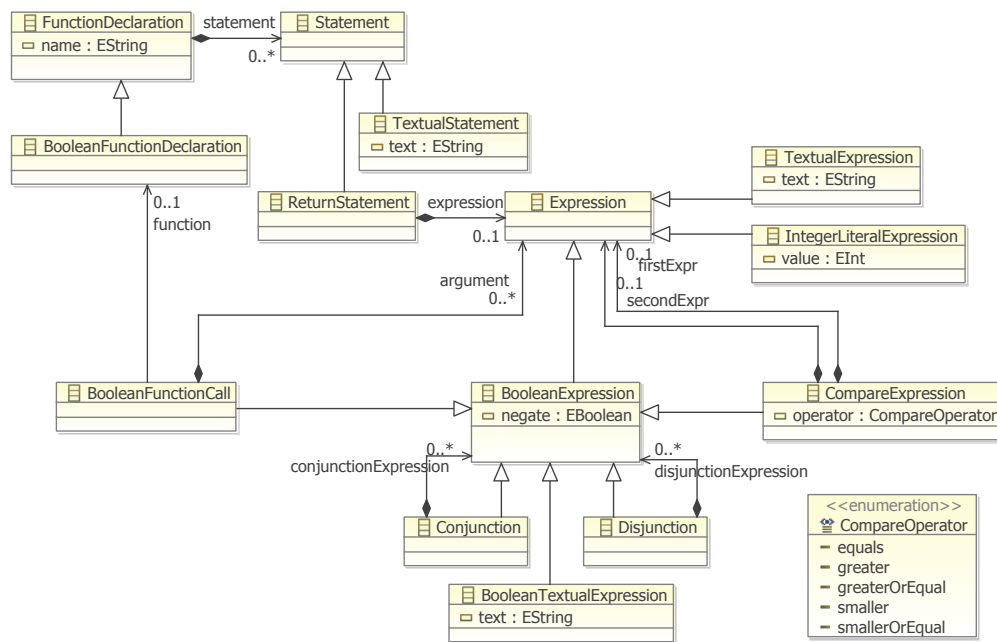


Figure A.2: An ECore meta-model for UPPAAL TIGA (functions, statements, expressions)

A.2 MSD specifications in UML

This section illustrates how an MSD specification is represented using UML2 and a profile. The MSD profile used here is a modified and extended variant of the profile created by Maoz in the context of the S2A project¹ [MH06]. In the following, first the stereotypes used in the MSD profile are presented. Second, the abstract syntax of an MSD specification in UML is illustrated by object diagrams. This shall help to understand the definition of the mapping from MSDs to the system of TGAs in Chap. B. For general information on UML profiles and the UML meta-classes, see the UML2 specification [UML09].

The diagram in Fig. A.3 shows the stereotypes and data types which extend UML for modeling MSD specifications. First, there is the stereotype `MSDSpec`

¹<http://www.wisdom.weizmann.ac.il/~maozs/s2a/>

ification, which extends the UML meta-class `Package`. By its attribute `specificationKind`, an MSD specification can be marked as a timed specification or an untimed specification. Second, there is the stereotype `SpecificationPart`, which extends the UML meta-class `Property`. Specification parts model the environment and system objects in the MSD specification. To distinguish environment and system objects, the `SpecificationPart` stereotype has the attribute `partKind`. This attribute is typed over the enumeration `PartKind`, which defines two literals “Environment” and “System”. Third, the profile contains the stereotype `ModalMessage`, which adds attributes for the temperature and execution kind to messages. The temperature attribute is typed over the enumeration `TemperatureKind`, which defines the two literals “cold” and “hot”. The execution kind attribute is typed over the enumeration `ExecutionKind`, which defines the two literals “monitor” and “execute”. Each message in an MSD specification must be a modal message, which is expressed by the “required”-annotation on the stereotype extension arrow. Forth, the profile contains the stereotype `Condition`, which extends the meta-class `StateInvariant`, and, like the modal message, defines an attribute `temperature`. Conditions are state invariants with the condition stereotype applied, assignments are state invariants with no stereotype applied. To syntactically distinguish time conditions and assignments which specify clock resets, there are the stereotypes `TimeCondition` and `ClockReset`.

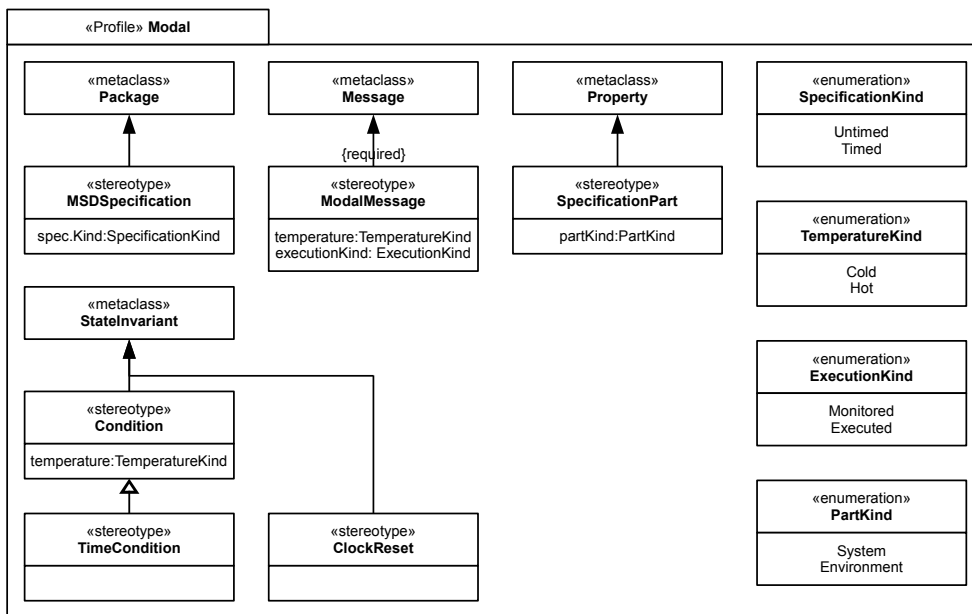


Figure A.3: The UML-Profile for MSD-Specifications

The object diagrams in Fig. A.4, A.5 and A.6 illustrate the abstract syntax of part of the MSD specification Drive onto track section modeled in UML and by using the above profile. The first two object diagrams show part of the MSD RequestEnterAtEndOfTrackSection and the corresponding collaboration as shown in in Fig. 4.2. The object diagram in Fig. A.6 shows how the time condition

of the MSD `ReplyOfSwitchControlNotTooEarly` in Fig. 4.8. is represented in the abstract syntax.

Figure A.4 shows the abstract syntax of a part of the MSD `RequestEnterAtEndOfTrackSection`, which is part of the MSD specification `Drive onto track section`. The root object is a package to which the stereotype `MSDSpecification` is applied. An MSD specification contains a collaboration, which in turn contains properties that represent the environment and system objects of the MSD specification. The stereotype `SpecificationPart` is applied to the properties. The attribute `partKind` determines whether a the property represents as system object or an environment object. The object diagram furthermore shows that the properties are typed over the classes contained in the package. In addition to the properties, the collaboration contains interactions. The object diagram shows the root object of the MSD `RequestEnterAtEndOfTrackSection` with its three lifelines (see Fig. 4.2). The object diagram also shows how the lifelines refer to the properties in the collaboration.

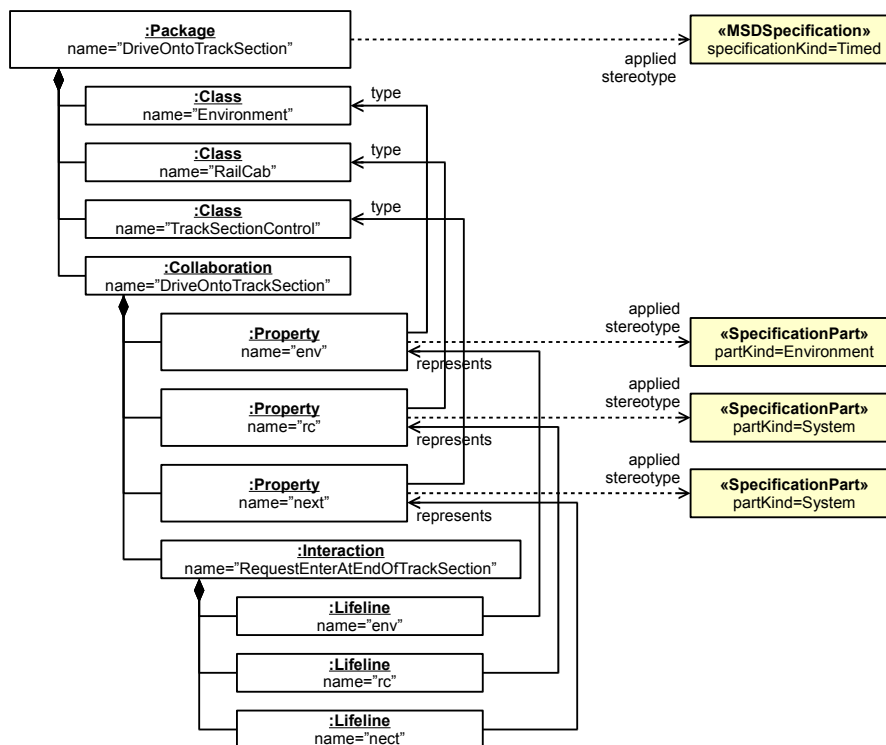


Figure A.4: The abstract syntax of an MSD specification: packages, classes, collaborations, parts, interactions, and lifelines

Figure A.5 shows further details of the abstract syntax; objects already appearing in Fig. A.4 are grayed-out. Besides the lifelines, the object diagram shows a message in the MSD. The message has the stereotype `ModalMessage` applied. The object diagram shows how the message is attached to the sending and receiving lifeline via message occurrence specifications. The diagram in particular shows that the message occurrence specifications are contained in

the interaction via the ordered reference fragments. The position in this list is denoted by the numbers in the brackets. This ordering reflects the order of events in a UML Sequence Diagram. Each message occurrence specification references a send operation event. For synchronous messages, the sending and receiving message occurrence specifications refer to the same event. The event in turn refers to an operation defined by the type class of the property referred to by the receiving lifeline. If there are multiple messages representing the same message kind in an MSD specification, there are multiple send operation events which refer to the same operation.

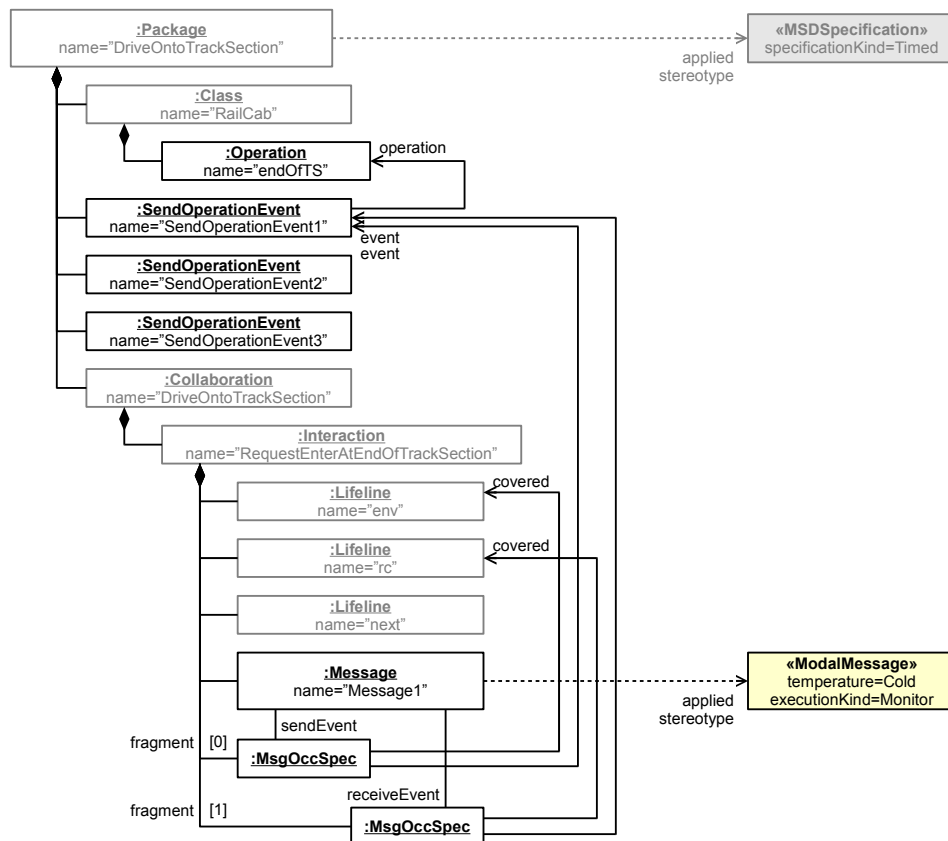


Figure A.5: The abstract syntax of an MSD specification: messages, occurrence specifications, events, and operations

The object diagram in Fig. A.6 shows the abstract syntax representation of a condition. In this case the time condition of the MSD ReplyOfSwitchControlNotTooEarly as shown in Fig. 4.8. Time conditions are state invariants with the stereotype TimeCondition applied. A state invariant can cover one or multiple lifelines that it references via the covered reference. A state invariant further can have a constraint with an expression that carries the expression string. Just like message occurrence specifications, state invariants are contained in the interaction via the ordered reference fragments. Fig. A.6 shows that the time condition of the MSD ReplyOfSwitchControlNotTooEarly in Fig. 4.8 is at position 5.

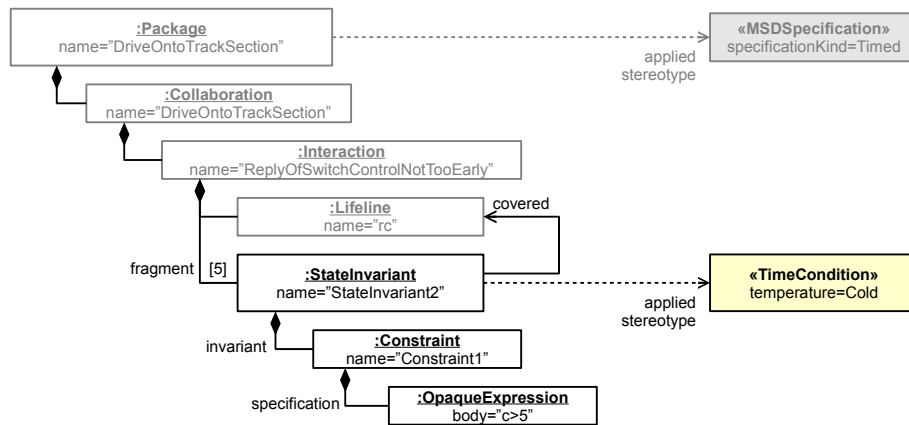


Figure A.6: The abstract syntax of an MSD specification: conditions and expressions

MSD-to-TGA TGG Transformation

This chapter presents the TGG-based mapping from MSD use case specifications to networks of timed automata as explained in Chap 4, called the *MSD-to-TGA mapping* in short. First, Sect. B.1 gives an overview of the TGG rules. Then Sect. B.2 lists a number of auxiliary operations, defined in OCL, that are used within the TGG rules. Last, Sect. B.3 lists all TGG rules in the MSD-to-TGA mapping.

The MSD-to-TGA mapping is part of SCENARIOTOOLS. Installing SCENARIOTOOLS or just the MSD-to-TGA transformation feature allows the reader to test the transformation or to inspect the TGG rules and example models in more detail. Installation instructions can be found on the following website:

<http://www.cs.upb.de/index.php?id=scenariotools>

B.1 TGG rule overview

The MSD-to-TGA TGG consists of one axiom and 30 TGG rules. Figure B.1 shows an overview that illustrates the rule generalization relationships as well as the dependencies between the rules. There is a dependency between rule if the application of a rule requires the application of one or several other rule applications. For example, in order to apply the rules `MSDSpecification` or `TimedMSDSpecification`, a binding of the axiom must exist; in order to apply the rule `Lifeline`, a binding of the rule `MSD` and `MSDSpecification` or `TimedMSDSpecification` must exist.

The TGG rules map an MSD use case specification to a network of TGA basically as follows: The axiom simply defines the correspondence between the root objects of the models, i.e., the package of the MSD specification and the NTA object (Fig. B.2). Then the rules `MSDSpecification` or `TimedMSDSpecification`

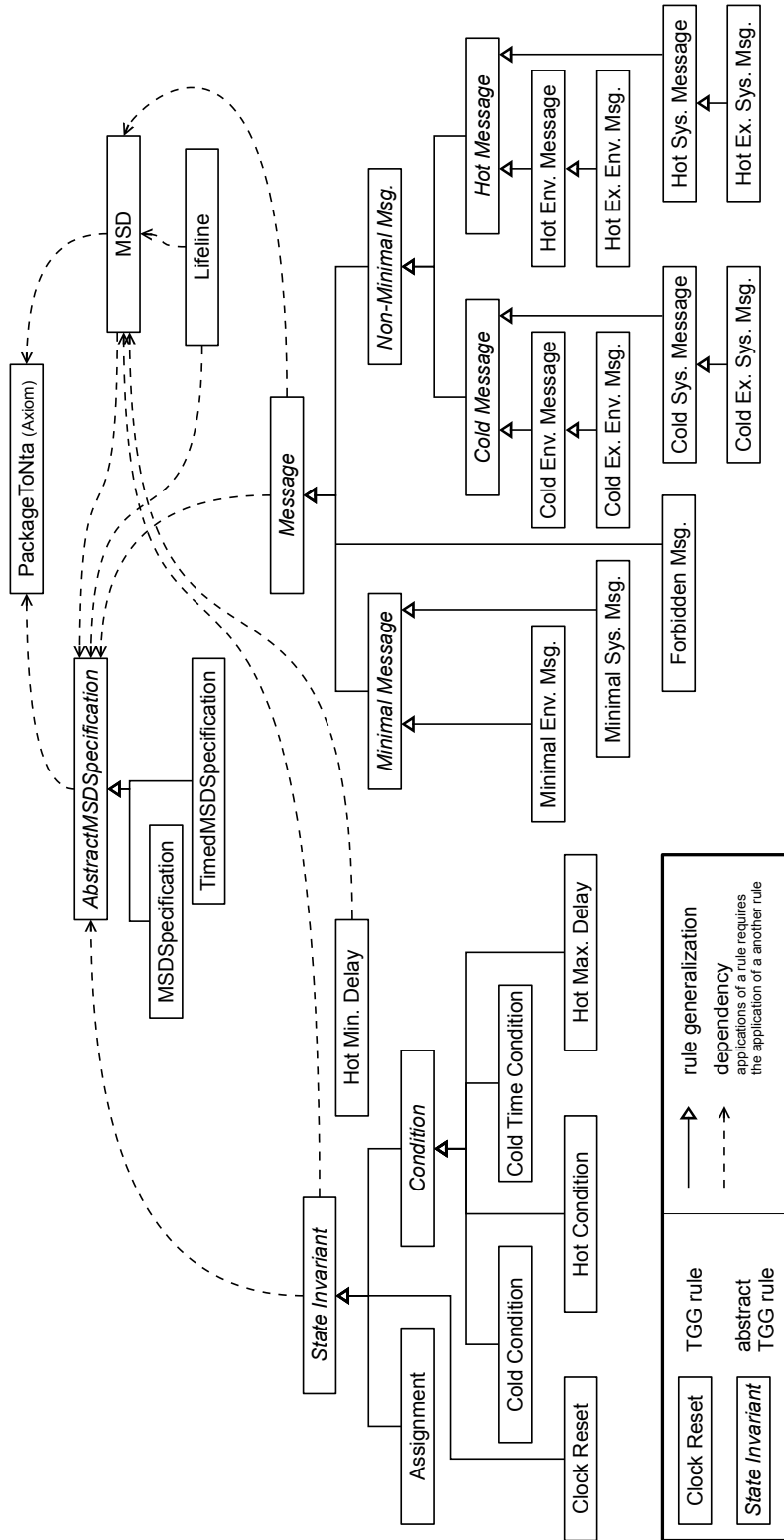


Figure B.1: An overview of the rule relationships in the TGG defining the MSD-to-TGA mapping

map the collaboration of the MSD specification and create the basic structure of various elements in the TGA model: the environment and system automaton templates, the global declarations along with a number of variables, function skeletons, and the system declaration, which defines instantiations of the environment and system automaton templates. Most of this mapping is defined in the abstract super-rule `AbstractMSDSpecification` (see Fig. B.3- B.6). The rules `MSDSpecification` or `TimedMSDSpecification` add aspects that are specific to untimed resp. timed MSD specifications (see Fig. B.7 and B.8).

Based on applications of `MSDSpecification` or `TimedMSDSpecification`, the rule `MSD` (Fig. B.9 to B.11) can be applied for each MSD in the MSD specification. The rule creates an MSD automaton template and its instantiation. Based on applications of the rule `MSD`, it is then possible to apply the rule `Lifeline` (Fig. B.12), which maps each lifeline of an MSD to a lifeline variable and parts of update and guard expressions of edges in the corresponding MSD automaton template, as well as parts of expressions in various function bodies. Based on applications of `MSD`, the rules for mapping various kinds of messages (Fig. B.13 to B.28) and state invariants, i.e., conditions, assignments, etc (Fig. B.29 to B.37), can be applied.

B.2 OCL attribute definitions

The following listing shows the contents of an OCL custom definitions file that is part of the mapping definition. It defines additional attributes and operations for UML elements that are used in constraints in the TGG rules. These additional attributes and operations mainly provide strings that are required in the TGA model. For example, the attribute `varName` produces the string that is used for the names of lifeline variables in the TGA model.

Listing B.1: Custom OCL definitions

```

package uml
context Lifeline
def: varName : String =
    self.interaction.name.concat('_')
    .concat(self.represents.name)
def: minimalEventOnLifeline : Boolean =
    self.interaction.fragment->at(1).covered
->includes(self) or
    self.interaction.fragment->at(2).covered
->includes(self)
def: allFragments: OrderedSet(InteractionFragment) =
    self.interaction.fragment
->select(f|f.covered->includes(self))
def: maxVarValue: Integer =
    self.allFragments->size()
+ (if self.minimalEventOnLifeline then 0 else 1 endif)
def: getPositionBefore(f:InteractionFragment) : Integer =
    self.allFragments->indexOf(f)
- (if self.minimalEventOnLifeline then 1 else 0 endif)
/* taken http://wiki.eclipse.org/OCLSnippets: */
def: toString(i:Integer) : String =

```

```

    OrderedSet{1000000, 10000, 1000, 100, 10, 1}->iterate(
      denominator : Integer;
      s : String = ''|
      let numberAsString : String = OrderedSet{
          '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
        }->at(i.div(denominator).mod(10) + 1)
      in
        if s='' and numberAsString = '0' then
          s
        else
          s.concat(numberAsString)
        endif
      )
  def: getPositionBeforeString(f: InteractionFragment)
      : String = self.toString(self.getPositionBefore(f))

context Message
  def: typeName : String =
    self.sendEvent
      .oclAsType(uuml::MessageOccurrenceSpecification).covered
      ->any(true).represents.name.concat('_')
      .concat(self.receiveEvent
        .oclAsType(uuml::MessageOccurrenceSpecification).covered
        ->any(true).represents.name).concat('_')
      .concat(self.sendEvent
        .oclAsType(uuml::MessageOccurrenceSpecification).event
        .oclAsType(uuml::SendOperationEvent).operation.name)
  def: increaseSendingReceivingLifelineExpr : String =
    self.sendEvent
      .oclAsType(uuml::MessageOccurrenceSpecification).covered
      ->any(true).varName.concat('+,□')
      .concat(self.receiveEvent
        .oclAsType(uuml::MessageOccurrenceSpecification).covered
        ->any(true).varName).concat('++')
  def: sendingMsgOccSpec : MessageOccurrenceSpecification =
    self.sendEvent
      .oclAsType(uuml::MessageOccurrenceSpecification)
  def: receivingMsgOccSpec : MessageOccurrenceSpecification =
    self.receiveEvent
      .oclAsType(uuml::MessageOccurrenceSpecification)
  def: sendingLifeline : Lifeline =
    self.sendingMsgOccSpec.covered->any(true)
  def: receivingLifeline : Lifeline =
    self.receivingMsgOccSpec.covered->any(true)
  def: sendingLocation : Integer =
    self.sendingLifeline.allFragments
      ->indexOf(self.sendingMsgOccSpec)
      - (if self.sendingLifeline.minimalEventOnLifeline
        then 1 else 0 endif)
  def: receivingLocation : Integer =
    self.receivingLifeline.allFragments
      ->indexOf(self.receivingMsgOccSpec)
      - (if self.receivingLifeline.minimalEventOnLifeline
        then 1 else 0 endif)

context Interaction
  def: increaseAllLifelineVariablesExpr : String =

```

```

    self.lifeline->iterate(ll : Lifeline; expr : String = '' |
    if expr = '' then ll.varName.concat('++')
    else expr.concat(',_').concat(ll.varName)
    .concat('++') endif)
def: allLifelineVariablesZeroExpr : String =
self.lifeline->iterate(ll : Lifeline; expr : String = '' |
if expr = '' then ll.varName.concat('_==_0')
else expr.concat('_and_')
.concat(ll.varName).concat('_==_0') endif)
def: resetAllLifelineVariablesExpr : String =
self.lifeline->iterate(ll : Lifeline; expr : String = '' |
if expr = '' then ll.varName.concat('_==_0')
else expr.concat(',_')
.concat(ll.varName).concat('_==_0') endif)

context StateInvariant
def: enabledExpr : String =
self.covered->iterate(ll : Lifeline; expr : String = '' |
if expr = '' then ll.varName.concat('_==_').
concat(ll.getPositionBeforeString(self))
else expr.concat('_and_').concat(ll.varName)
.concat('_==_')
.concat(ll.getPositionBeforeString(self))
endif)
def: increaseAllCoveredLifelineVariablesExpr : String =
self.covered->iterate(ll : Lifeline; expr : String = '' |
if expr = '' then ll.varName.concat('++')
else expr.concat(',_').concat(ll.varName)
.concat('++') endif)
def: resetAllCoveredLifelineVariablesExpr : String =
self.covered->iterate(ll : Lifeline; expr : String = '' |
if expr = '' then ll.varName.concat('_==_0')
else expr.concat(',_').concat(ll.varName)
.concat('_==_0') endif)

context OpaqueExpression
def: at(i:Integer) : String =
self.body->any(true).substring(i, i)
def: helperSequence : Sequence(Integer) =
Sequence{1..self.body->any(true).size()}
/* this is a little tricky, because OCL does not support
many string operations. Probably it would be better
to extend OCL: */
def: firstOccOfEquals : Integer = self.helperSequence
->iterate(counter : Integer; result : Integer = 0 |
if ((self.at(counter) = '=' or self.at(counter) = '_')
and result = 0) then counter else result endif)
/* Self checks whether the expression contains an '<': */
def: isMaximalDelay : Boolean = self.helperSequence
->iterate(counter : Integer; result : Integer = 0 |
if ((self.at(counter) = '<')
and result = 0) then counter else result endif) > 0
def: varName : String =
self.body->any(true)
.substring(1, self.firstOccOfEquals-1)

endpackage

```

B.3 TGG rules

This section lists the TGG rules in the MSD-to-TGA mapping as shown in the overview in Fig. B.1.

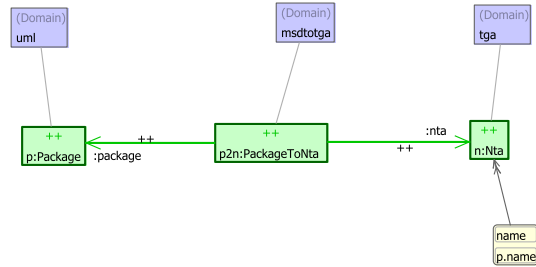


Figure B.2: The TGG Axiom PackageToNta

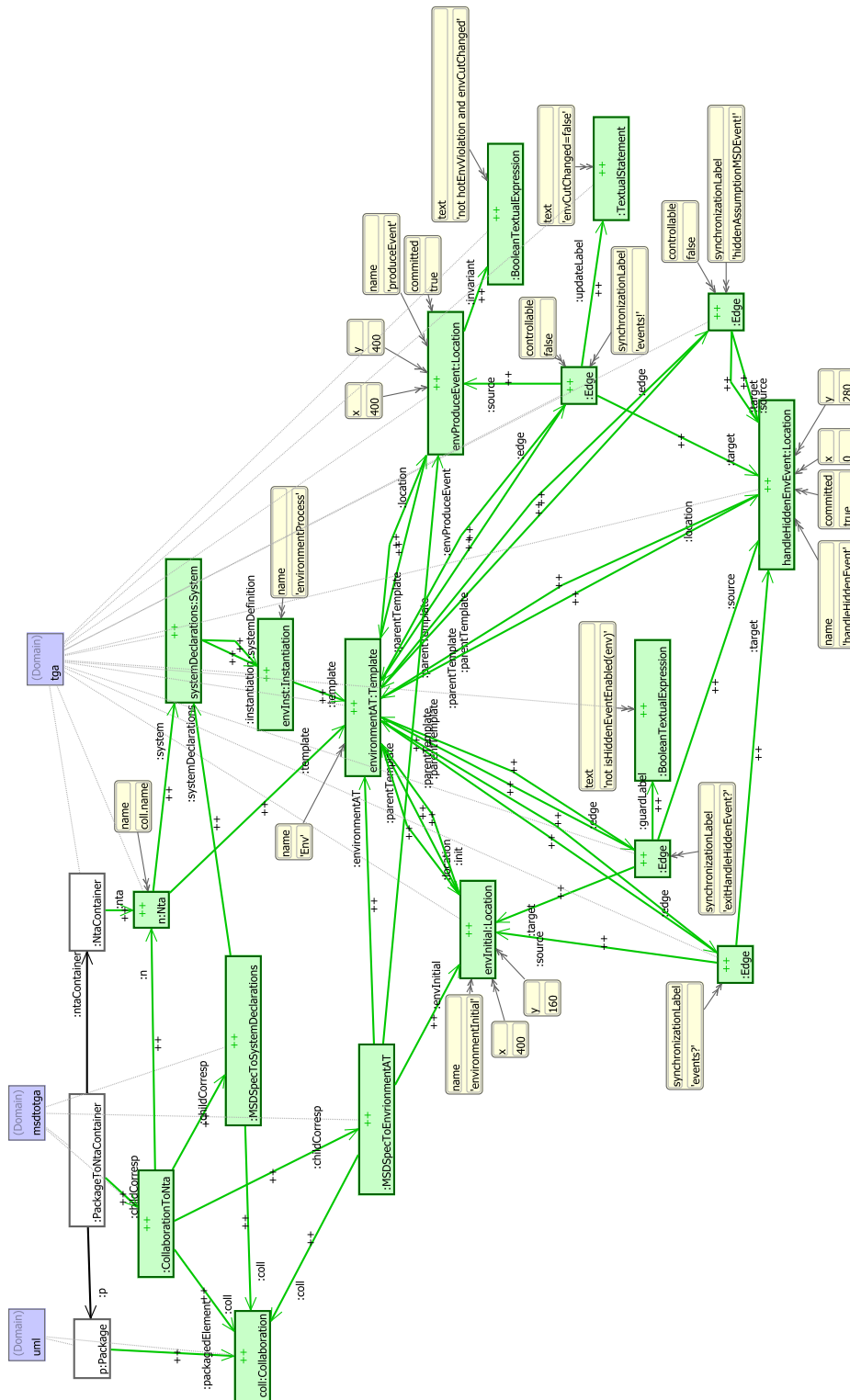


Figure B.3: Part of the TGG rule AbstractMSDSpecification (showing the environment automaton template created for an MSD specification as well as its instantiation)

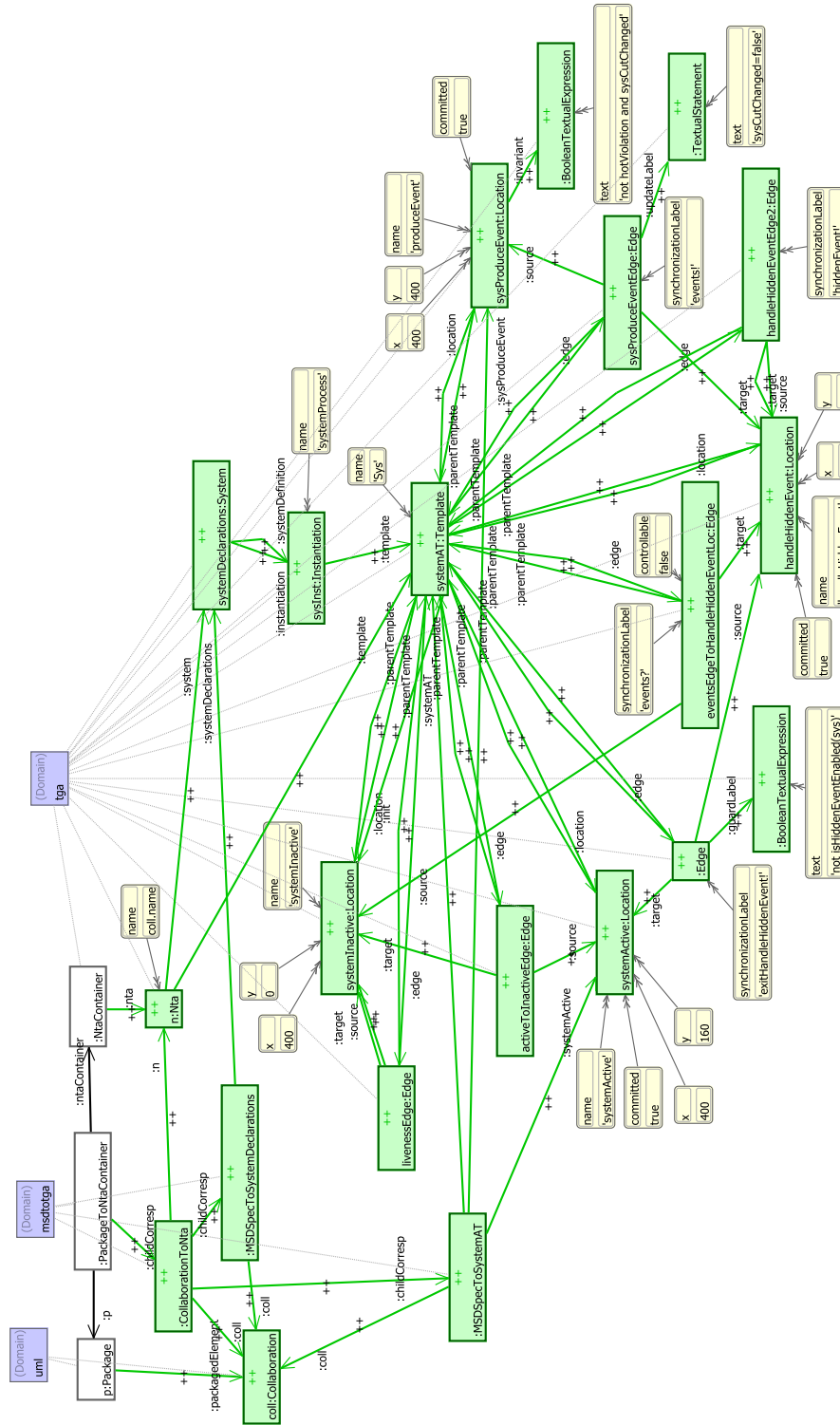


Figure B.4: Part of the TGG rule AbstractMSDSpecification (showing the system automaton template created for an MSD specification as well as its instantiation)

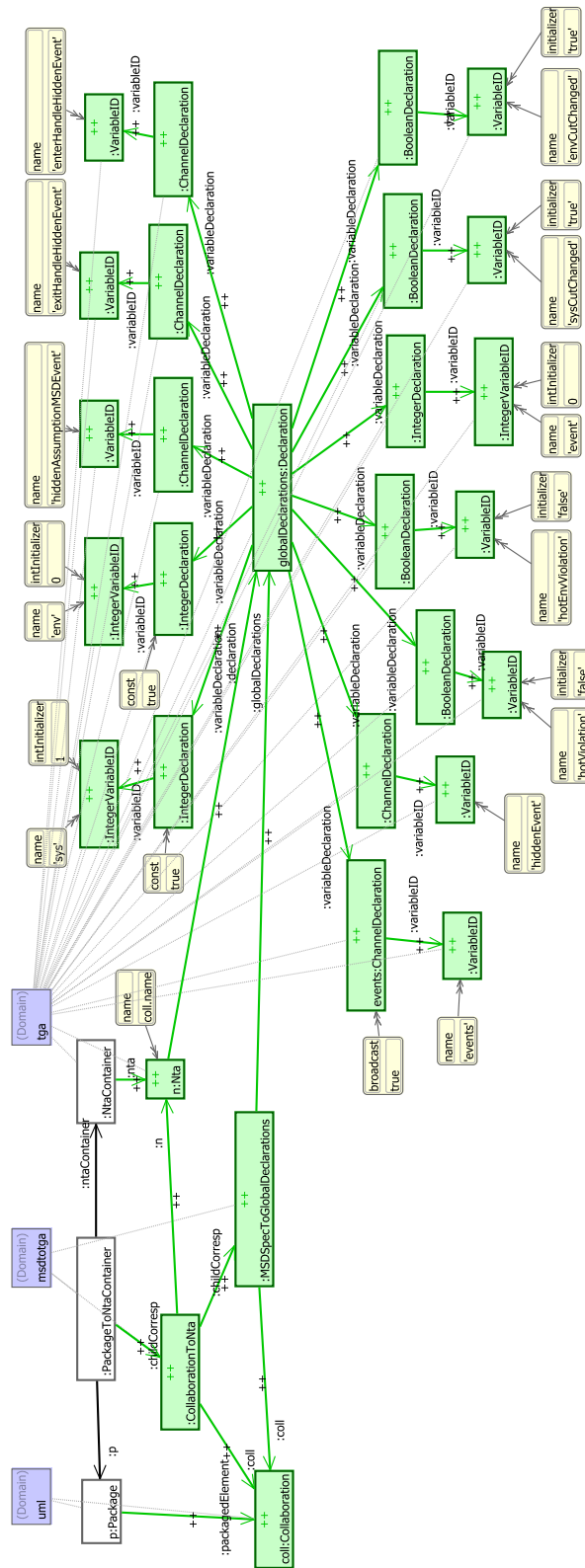


Figure B.5: Part of the TGG rule AbstractMSDSpecification (showing the global variable and channel declarations that are created for an MSD specification)

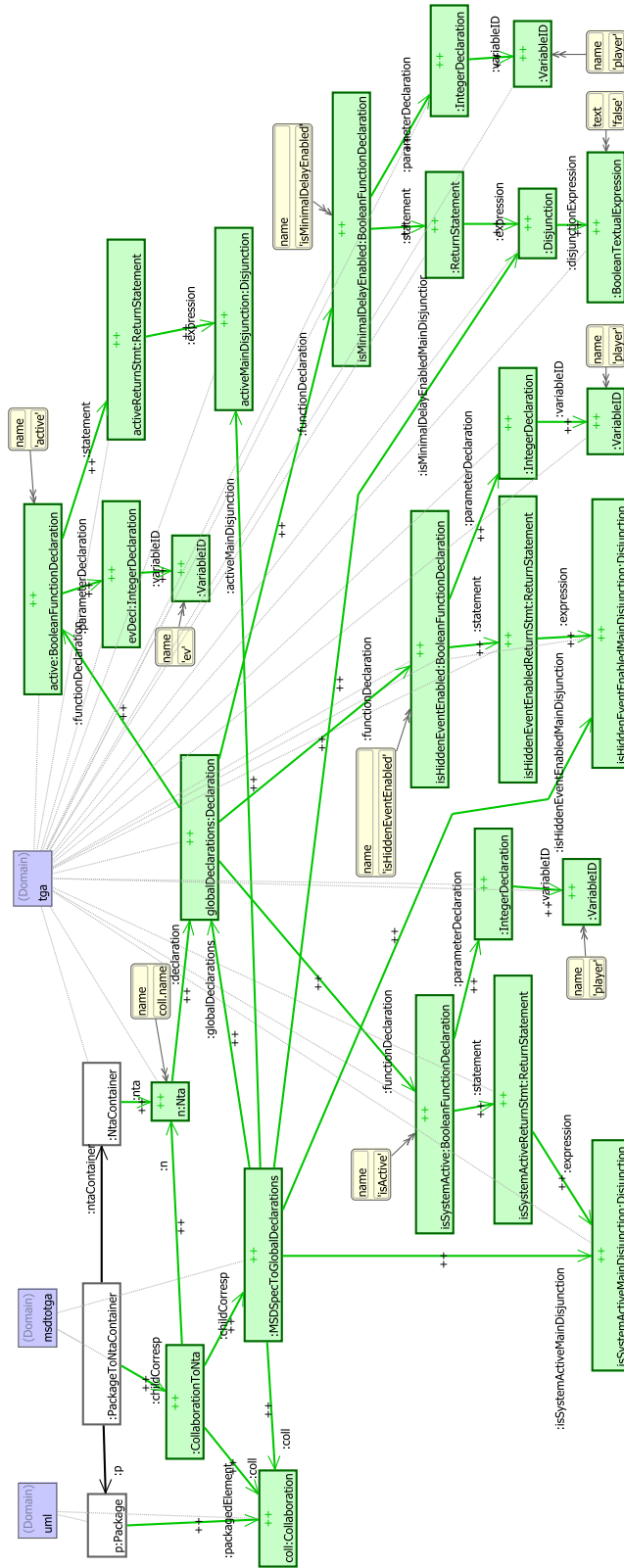


Figure B.6: Part of the TGG rule AbstractMSDSpecification (showing the global function declarations that are created for an MSD specification)

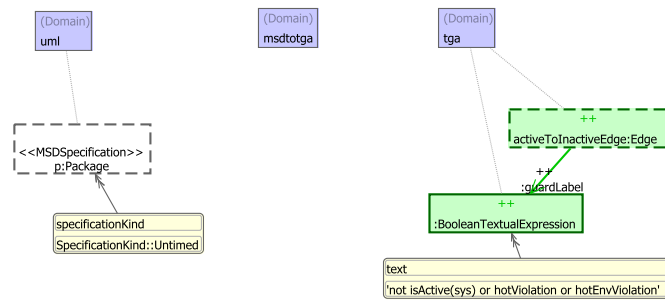


Figure B.7: The TGG rule MSDSpecification (refines AbstractMSDSpecification)

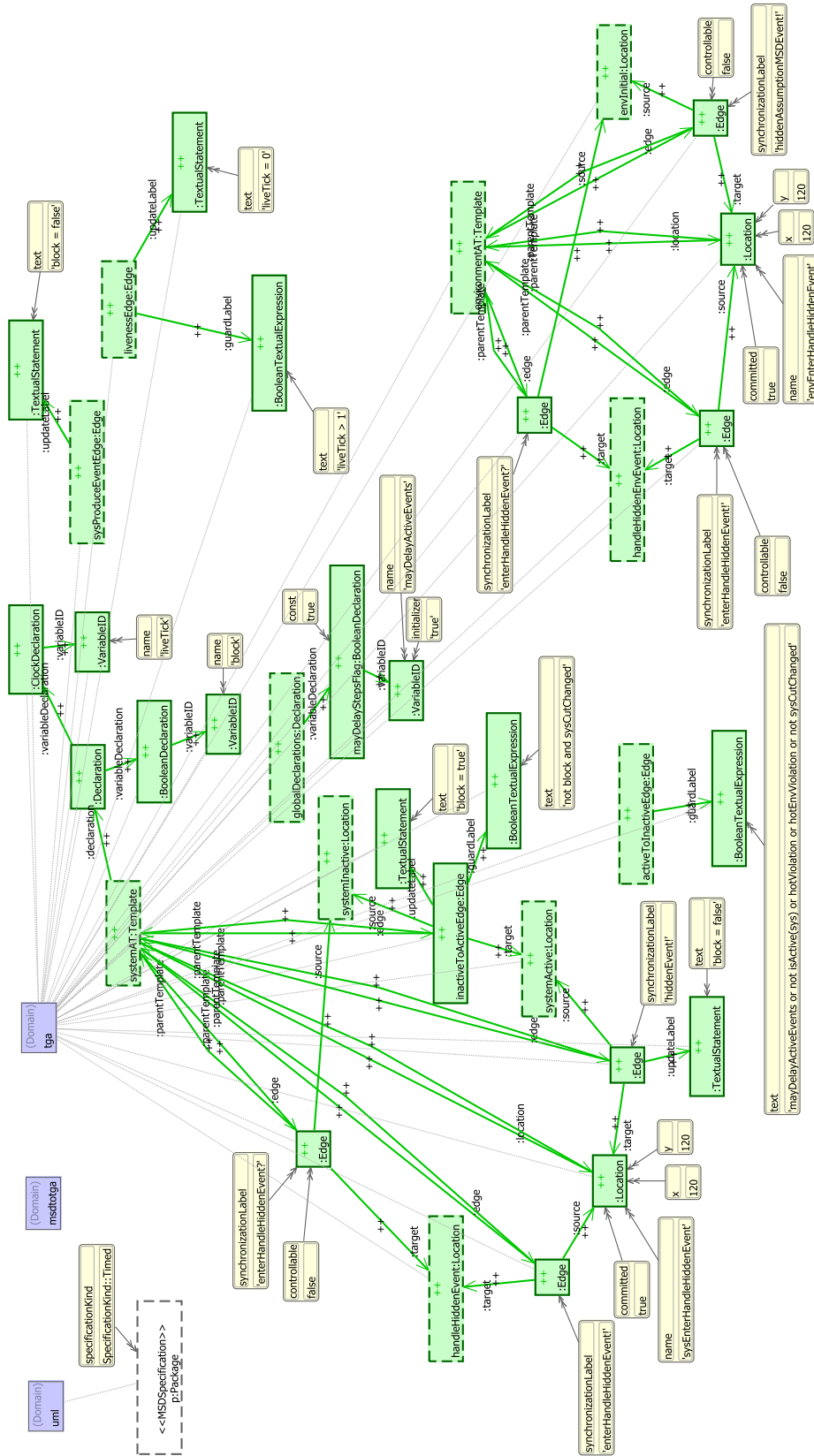


Figure B.8: The TGG rule `TimedMSDSpecification` (refines `AbstractMSDSpecification`)

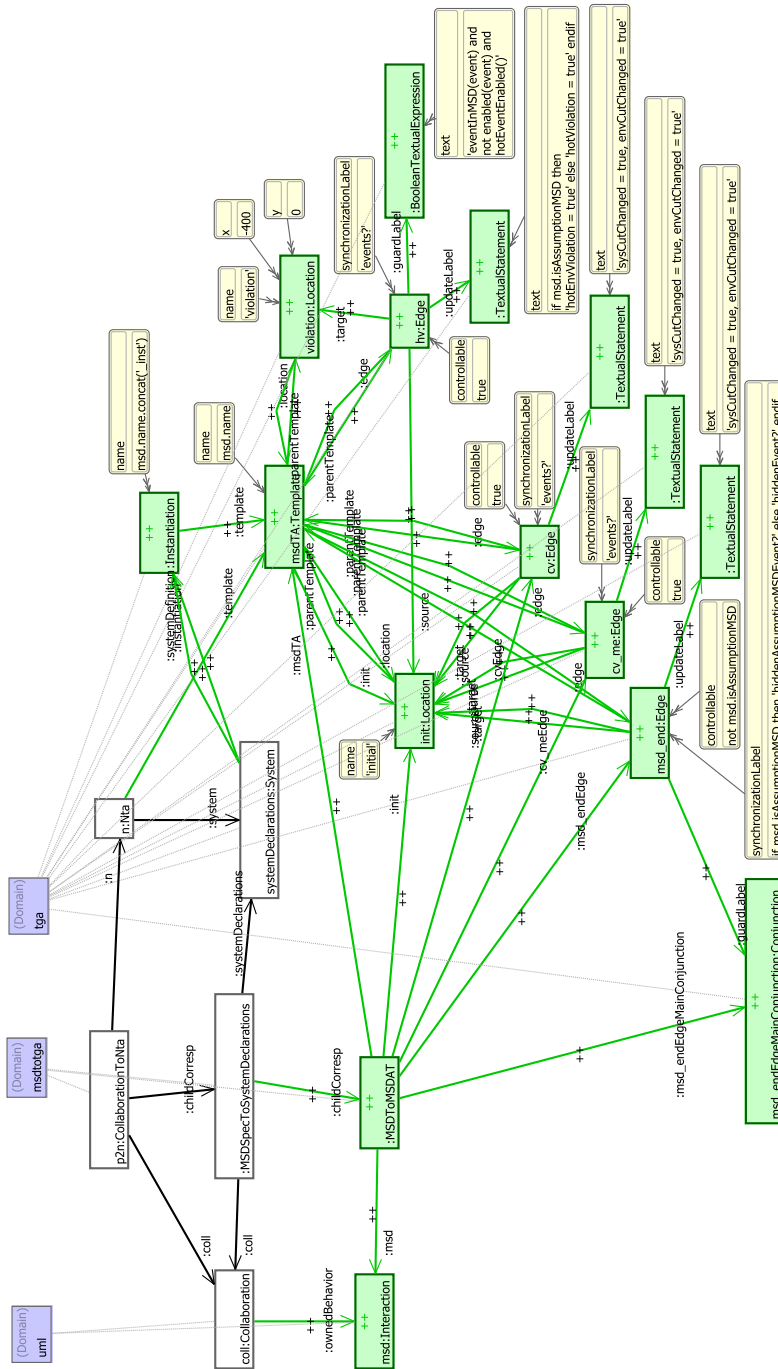


Figure B.9: Part of the TGG rule MSD (showing the MSD automaton template skeleton structure created for each MSD as well as the instantiation of the template)

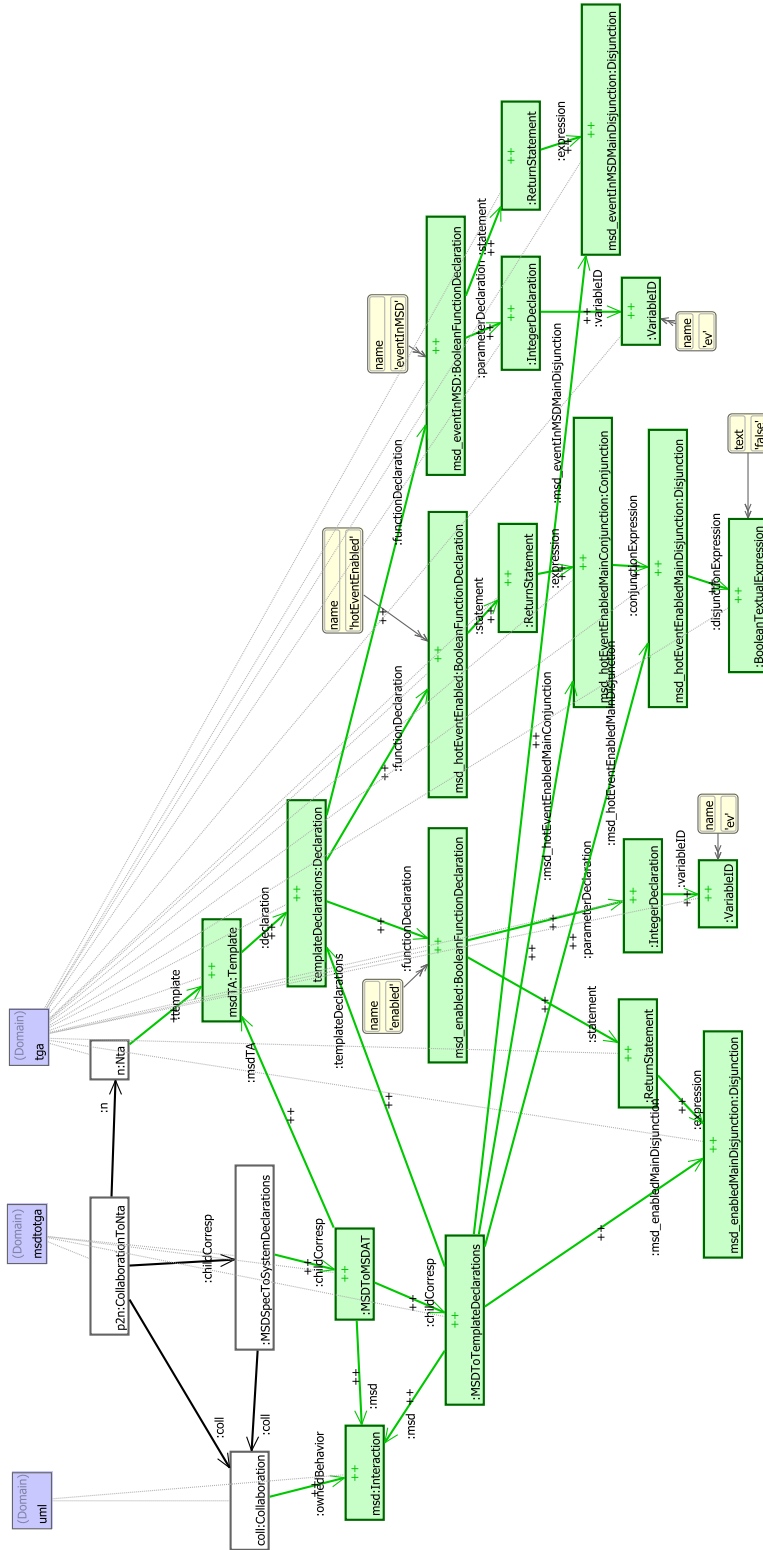


Figure B.10: Part of the TGG rule MSD (showing the template variable and function declarations created for an MSD automaton template)

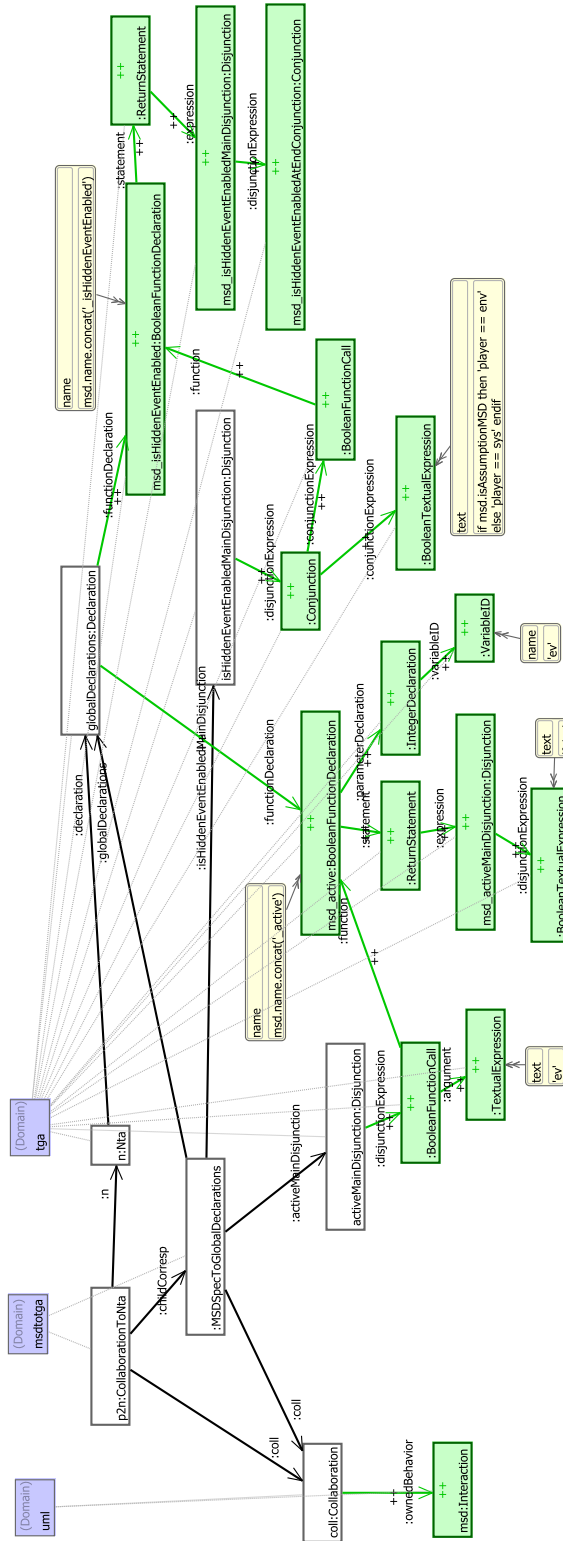


Figure B.11: Part of the TGG rule MSD (showing the global function declarations created for each MSD)

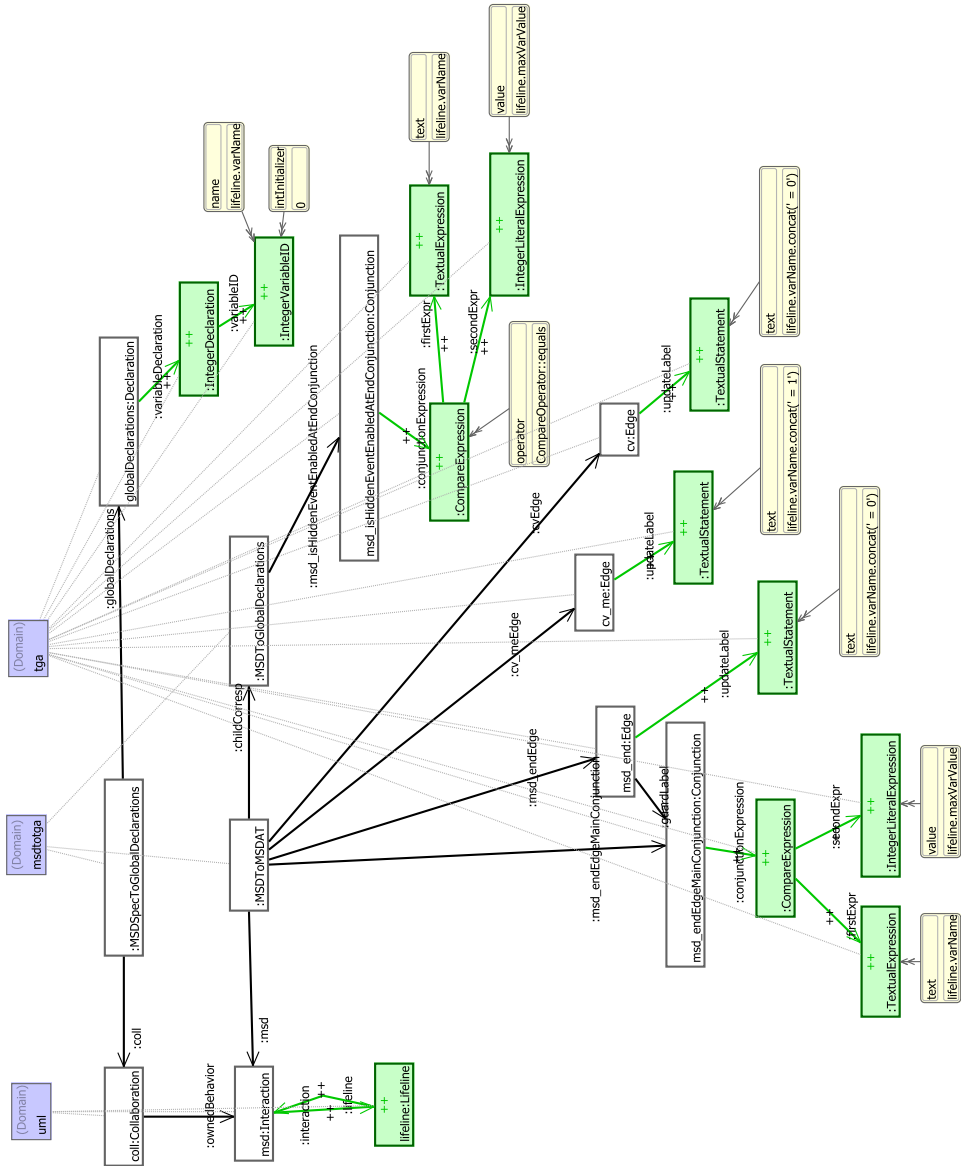


Figure B.12: The TGG rule Lifeline (mapping a lifeline to the declaration of a lifeline variable and to structures which represent parts of MSD automaton template edge update and guard expressions; furthermore mapped to part of the bool isHiddenEventEnabled(int player) function body)

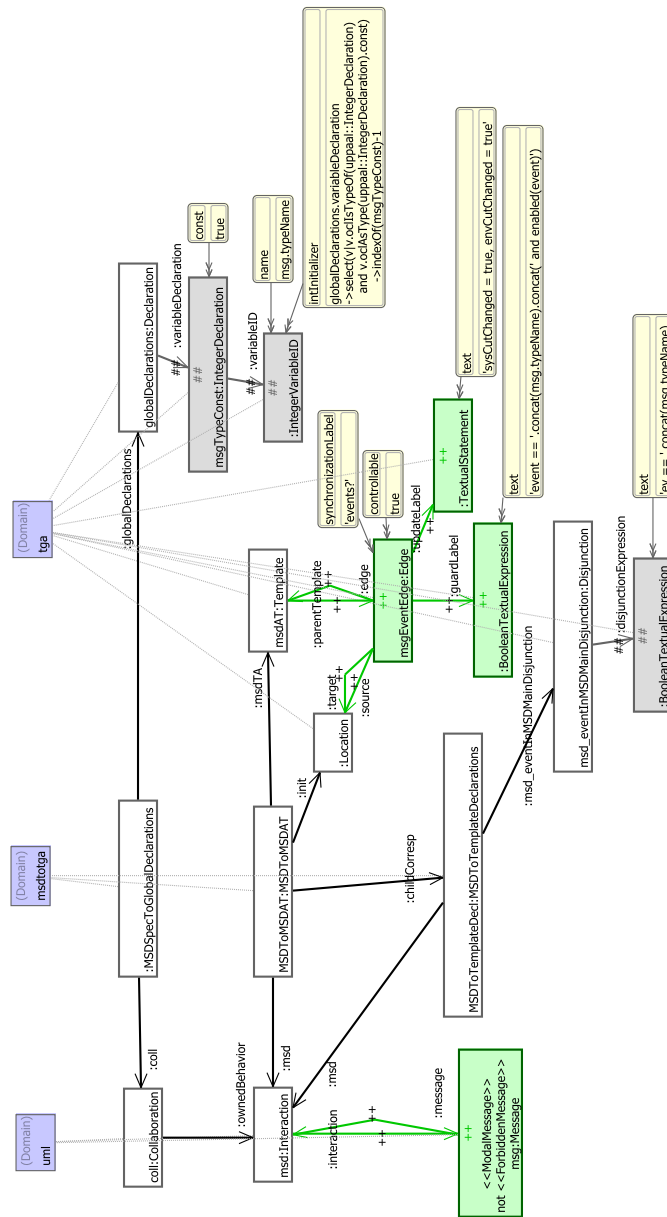


Figure B.13: The TGG rule Message (mapping a message in a TGG to a globally declared constant, an edge in the MSD automaton template, and part of an expression that is declared for the MSD automaton template)

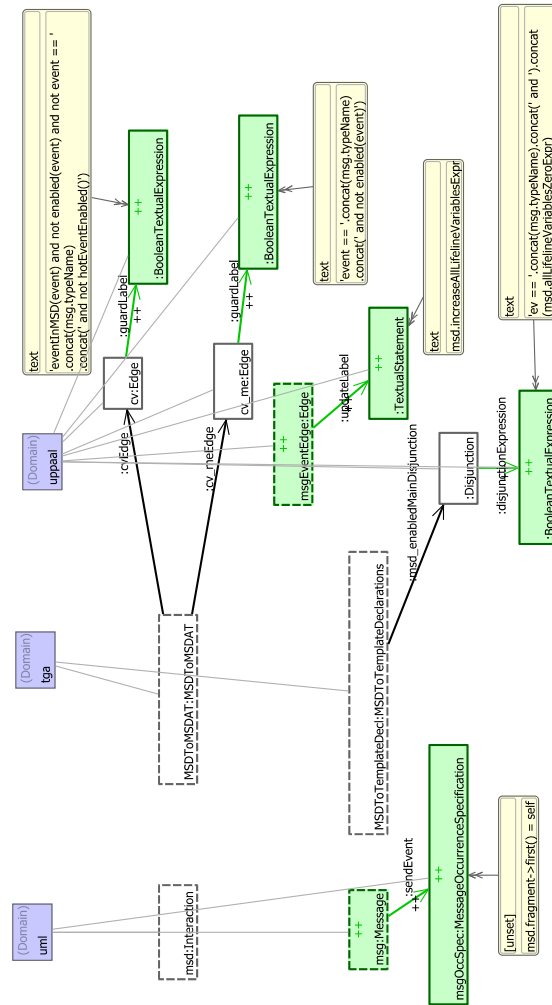


Figure B.14: The TGG rule `MinimalMessage` (mapping a minimal message to different parts of edge of edge update and guard labels in the MSD automaton template as well as part of `bool enabled (int ev)` function that is declared for the MSD automaton template, refines the rule `Message`)

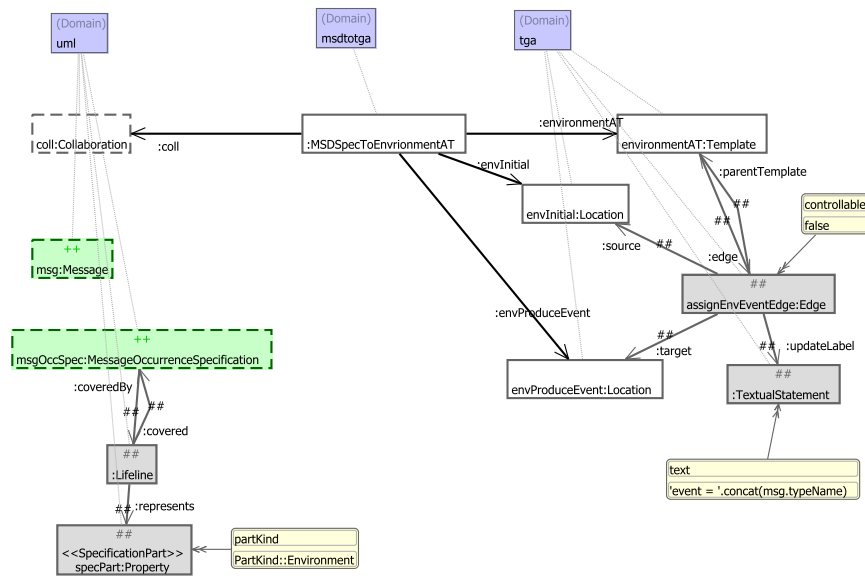


Figure B.15: The TGG rule MinimalEnvironmentMessage (mapping a minimal environment message to an edge in the environment automaton template, refines the rule MinimalMessage)

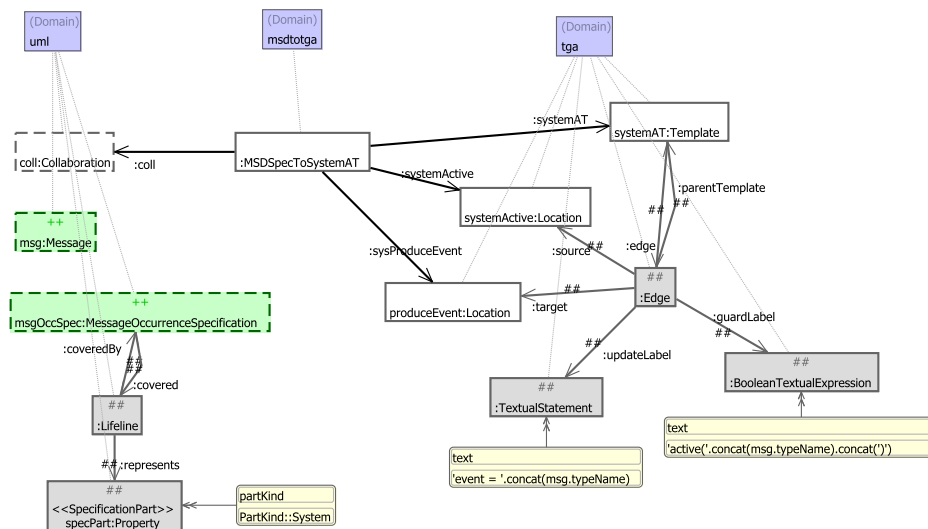


Figure B.16: The TGG rule MinimalSystemMessage (mapping a minimal system message to an edge in the system automaton template, refines the rule MinimalMessage)

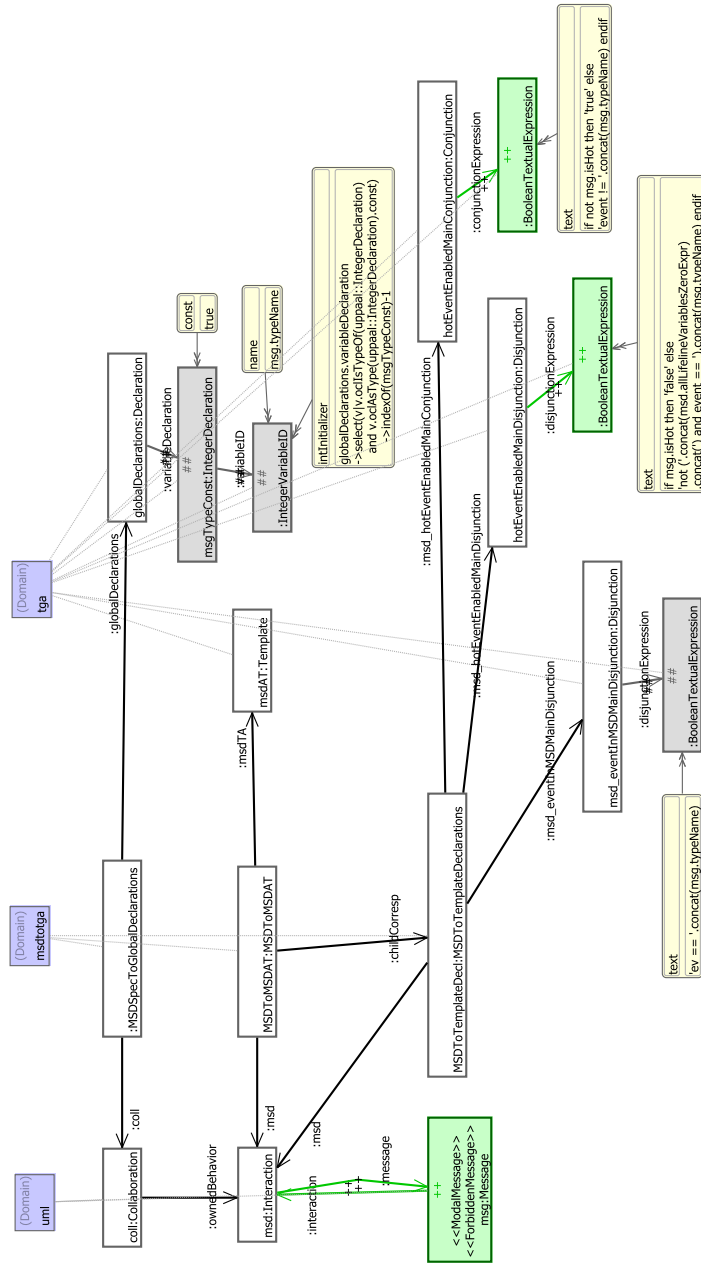


Figure B.17: The TGG rule **ForbiddenMessage** (mapping a forbidden message to parts of various function body expressions, refines the rule **Message**)

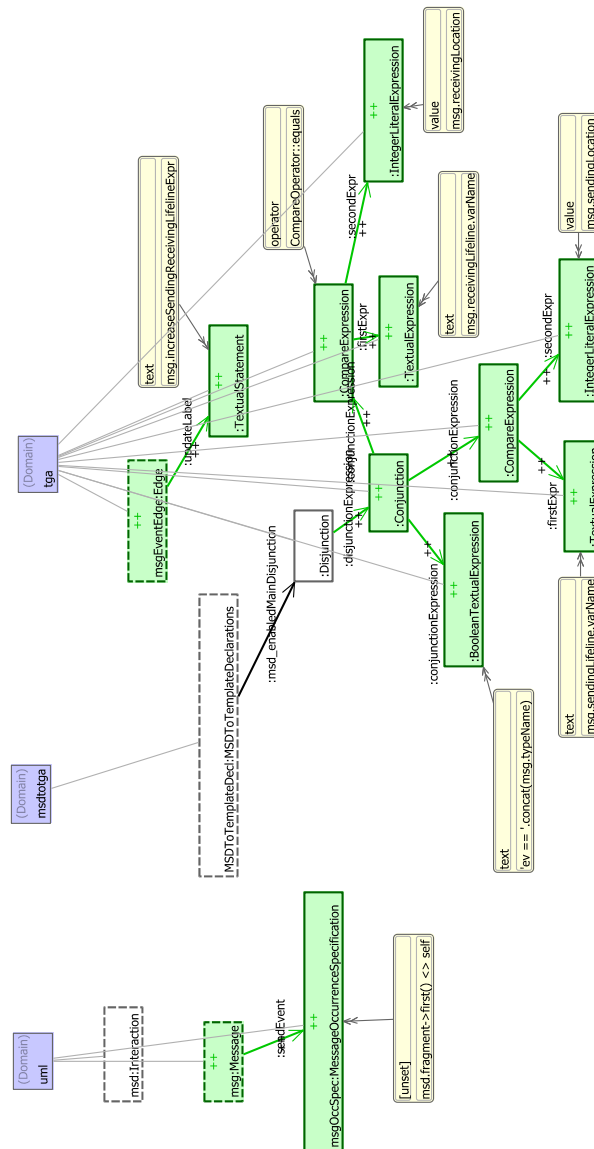


Figure B.18: The TGG rule `NonMinimalMessage` (mapping a non-minimal message to an edge update label in the MSD automaton template as well as part of a body expression of the function `bool enabled(int ev)` in the MSD automaton template, refines the rule `Message`)

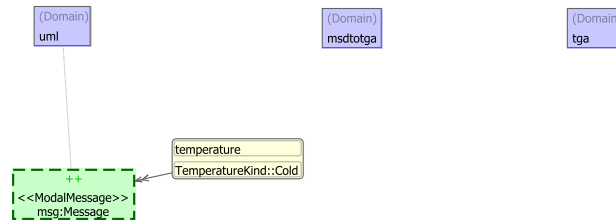


Figure B.19: The TGG rule `ColdMessage` (mapping a cold message, refines the rule `NonMinimalMessage`)

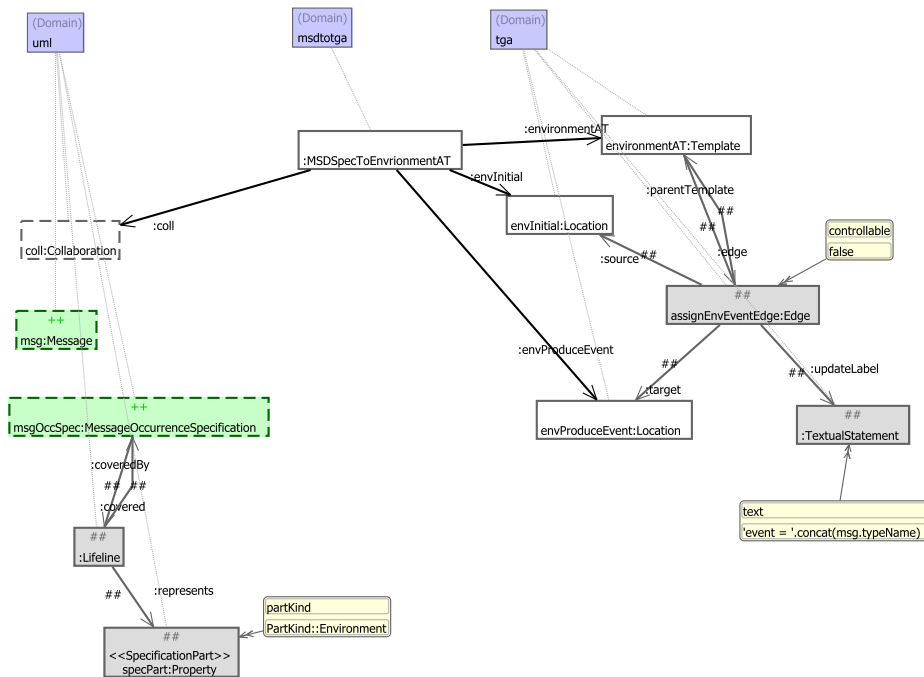


Figure B.20: The TGG rule `ColdEnvironmentMessage` (mapping a cold (monitored) environment message to the corresponding edge in the MSD automaton template, refines the rule `ColdMessage`)

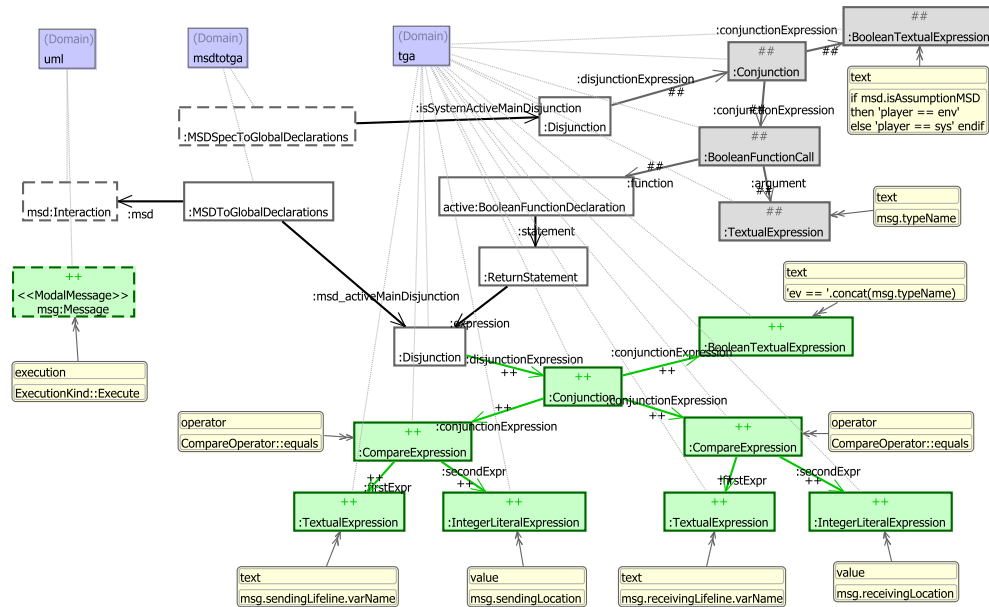


Figure B.21: The TGG rule ColdExecutedEnvironmentMessage (mapping a cold executed environment message to expression parts in global functions, refines the rule ColdEnvironmentMessage)

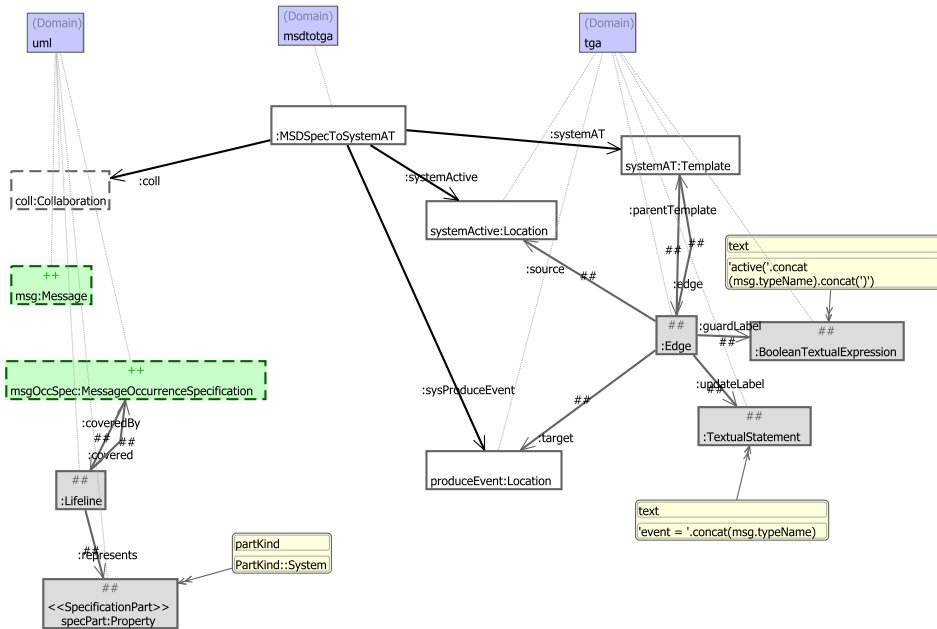


Figure B.22: The TGG rule ColdSystemMessage (mapping a cold (monitored) system message to the corresponding edge in the MSD automaton template, refines the rule ColdMessage)

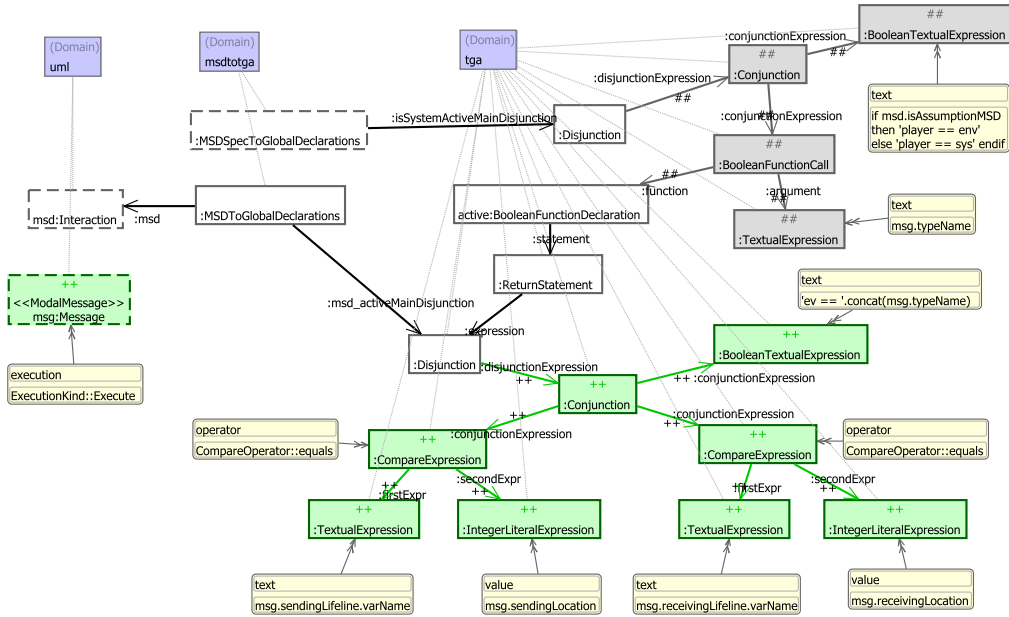


Figure B.23: The TGG rule ColdExecutedSystemMessage (mapping a cold executed system message to expression parts in global functions, refines the rule ColdSystemMessage)

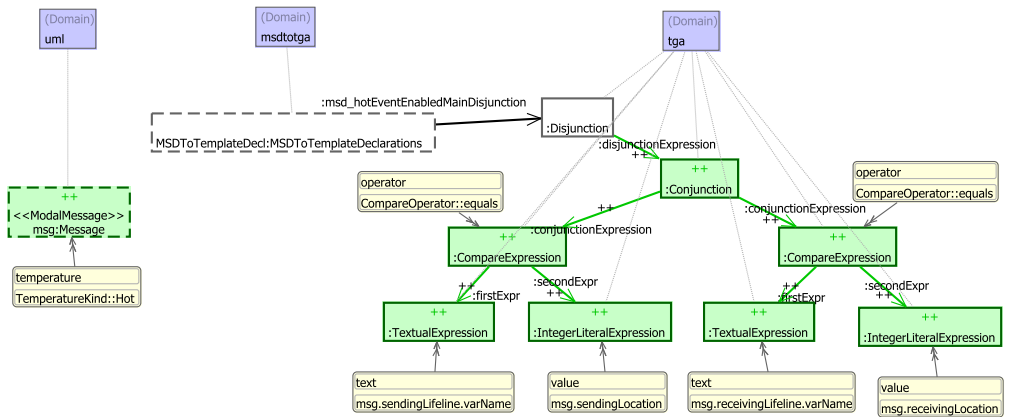


Figure B.24: The TGG rule HotMessage (mapping a hot message, refines the rule NonMinimalMessage)

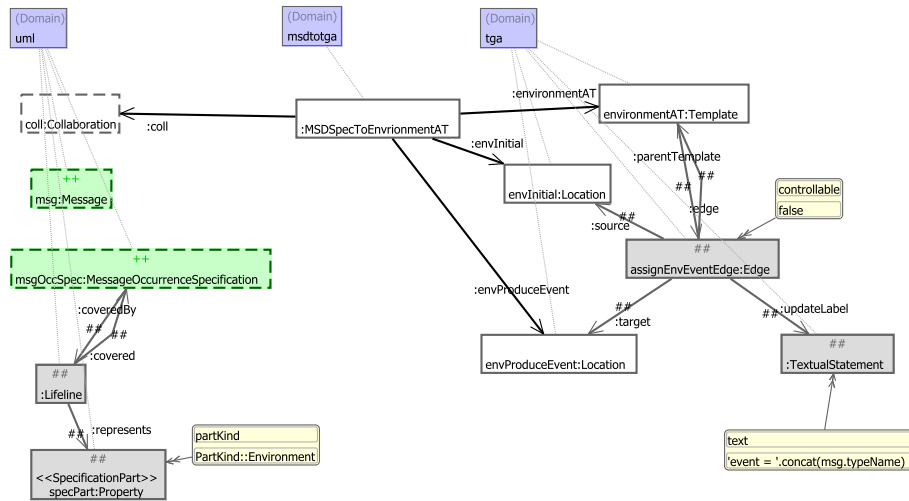


Figure B.25: The TGG rule `HotEnvironmentMessage` (mapping a hot (monitored) environment message to the corresponding edge in the MSD automaton template, refines the rule `HotMessage`)

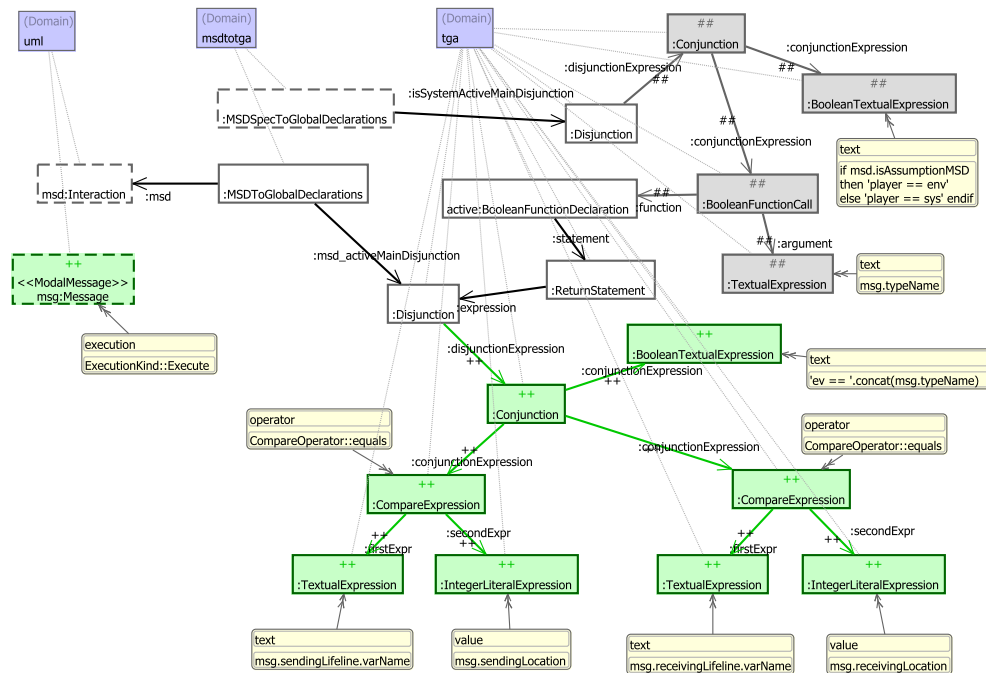


Figure B.26: The TGG rule `HotExecutedEnvironmentMessage` (mapping a hot executed environment message to expression parts in global functions, refines the rule `HotEnvironmentMessage`)

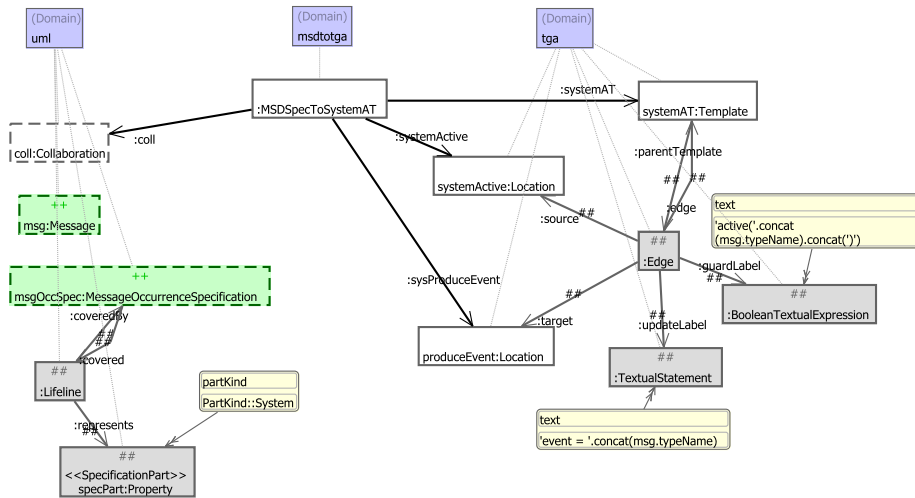


Figure B.27: The TGG rule `HotSystemMessage` (mapping a hot (monitored) system message to the corresponding edge in the MSD automaton template, refines the rule `HotMessage`)

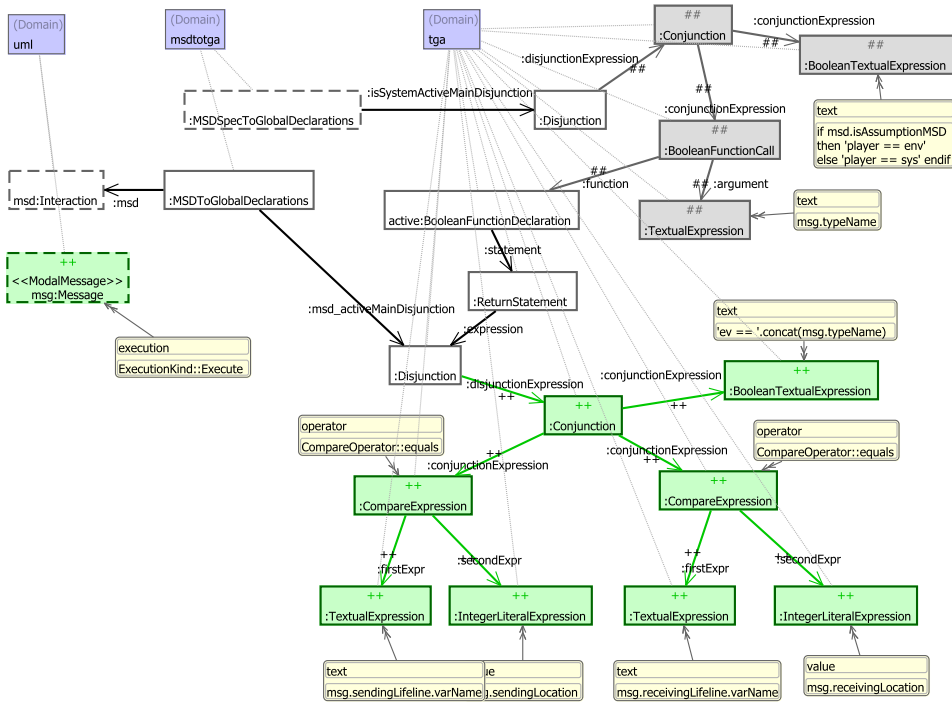


Figure B.28: The TGG rule `HotExecutedSystemMessage` (mapping a hot executed system message to expression parts in global functions, refines the rule `HotSystemMessage`)

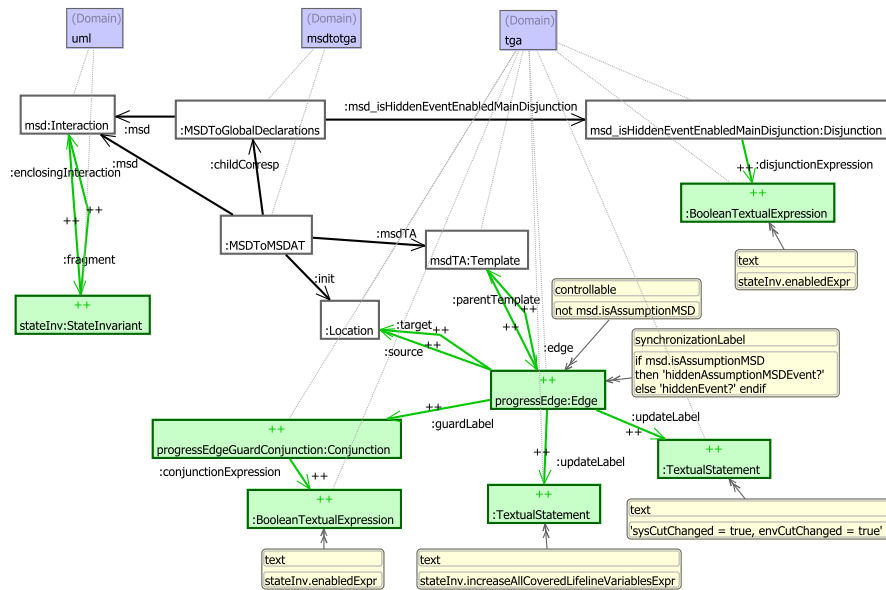


Figure B.29: The TGG rule StateInvariant (mapping a state invariant to an edge in the MSD automaton template and parts of global functions)

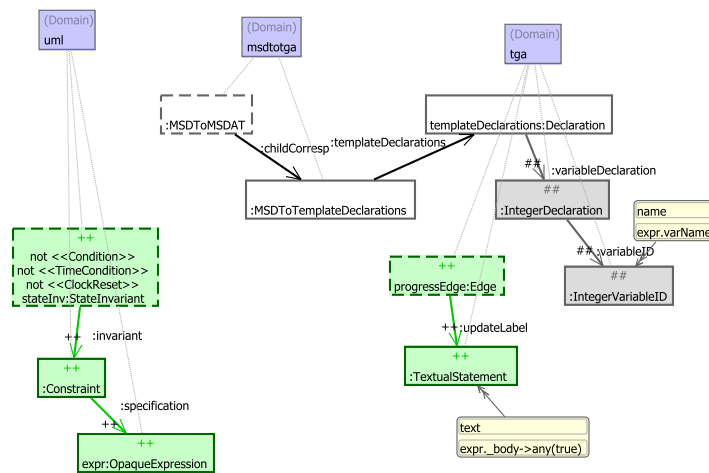


Figure B.30: The TGG rule Assignment (mapping an assignment to a variable declaration in the corresponding MSD automaton template and adding an update expression to the according edge in the MSD automaton template, refines the rule StateInvariant)

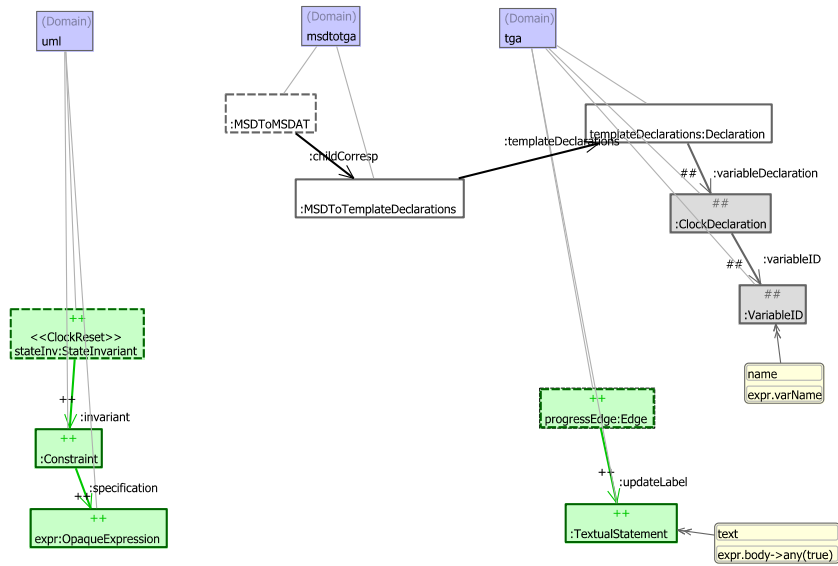


Figure B.31: The TGG rule ClockReset (mapping a clock reset to a clock declaration in the corresponding MSD automaton template and adding an update expression to the according edge in the MSD automaton template, refines the rule StateInvariant)

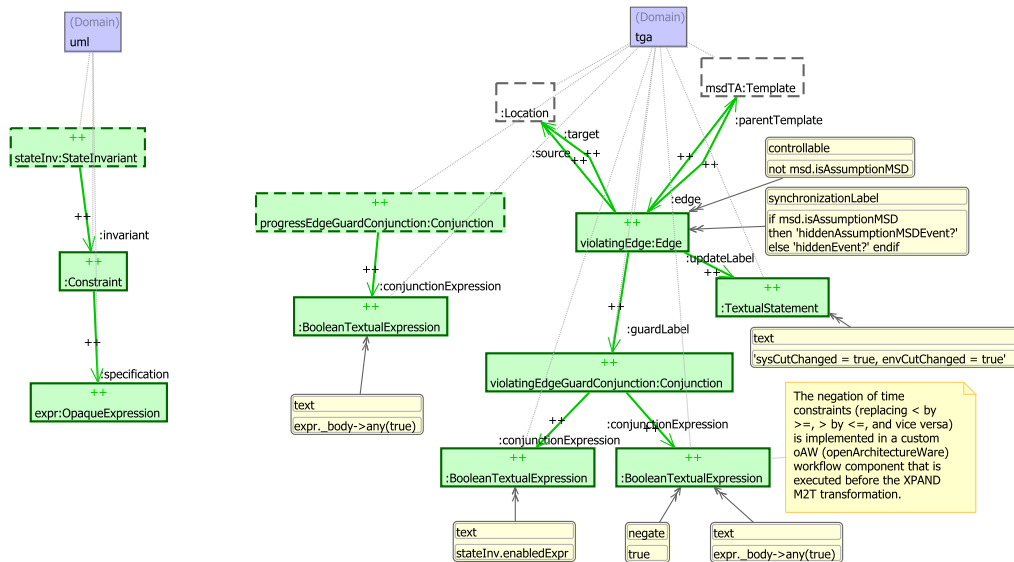


Figure B.32: The TGG rule Condition (mapping a condition to an additional part of the corresponding edge's guard label as well as an additional edge that represents the violation of the condition, refines the rule StateInvariant)

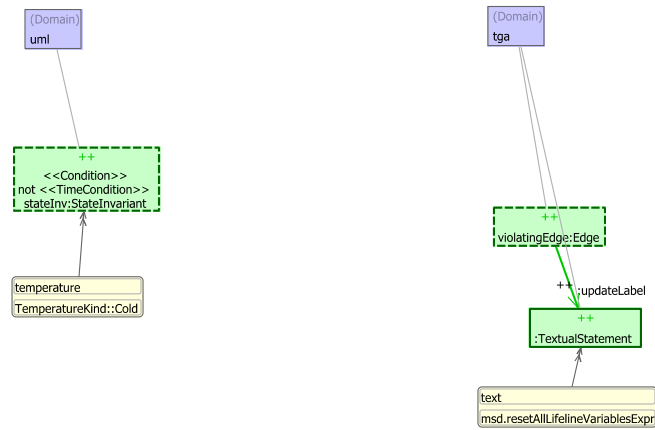


Figure B.33: TGG Rule ColdCondition (mapping a cold condition to an update label statement of the corresponding violating edge, refines the rule Condition)

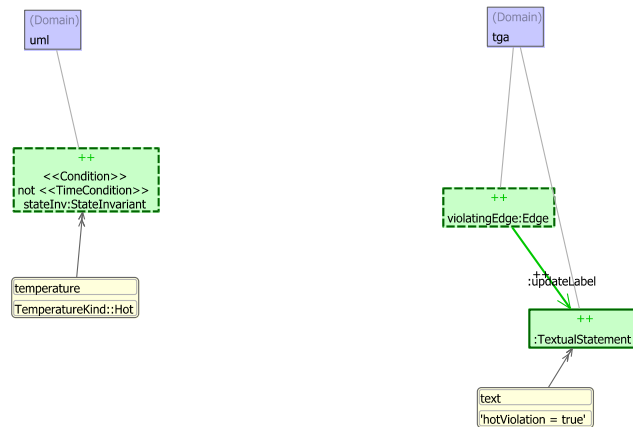


Figure B.34: The TGG rule HotCondition (mapping a hot condition to an update label statement of the corresponding violating edge, refines the rule Condition)

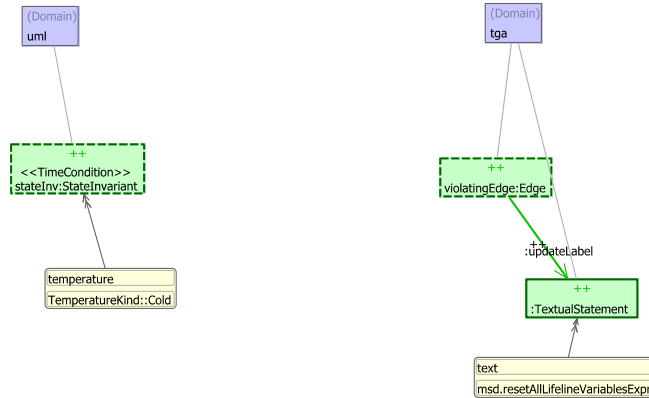


Figure B.35: The TGG rule `ColdTimeCondition` (mapping a cold time condition to an update label statement of the corresponding violating edge, refines the rule `Condition`)

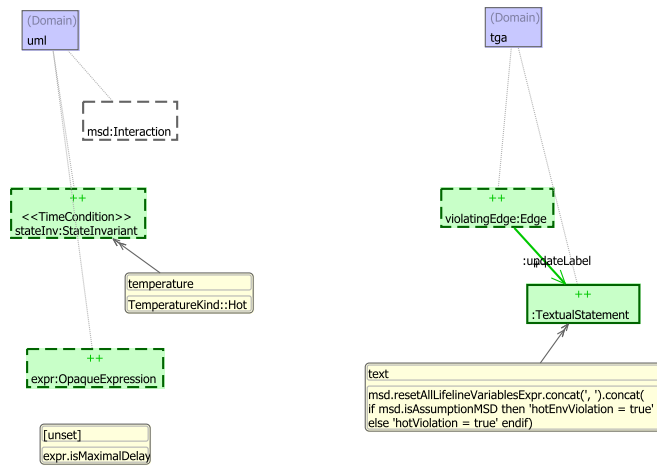


Figure B.36: The TGG rule `HotMaximalDelay` (mapping a hot maximal delay to an update label statement of the corresponding violating edge, refines the rule `Condition`)

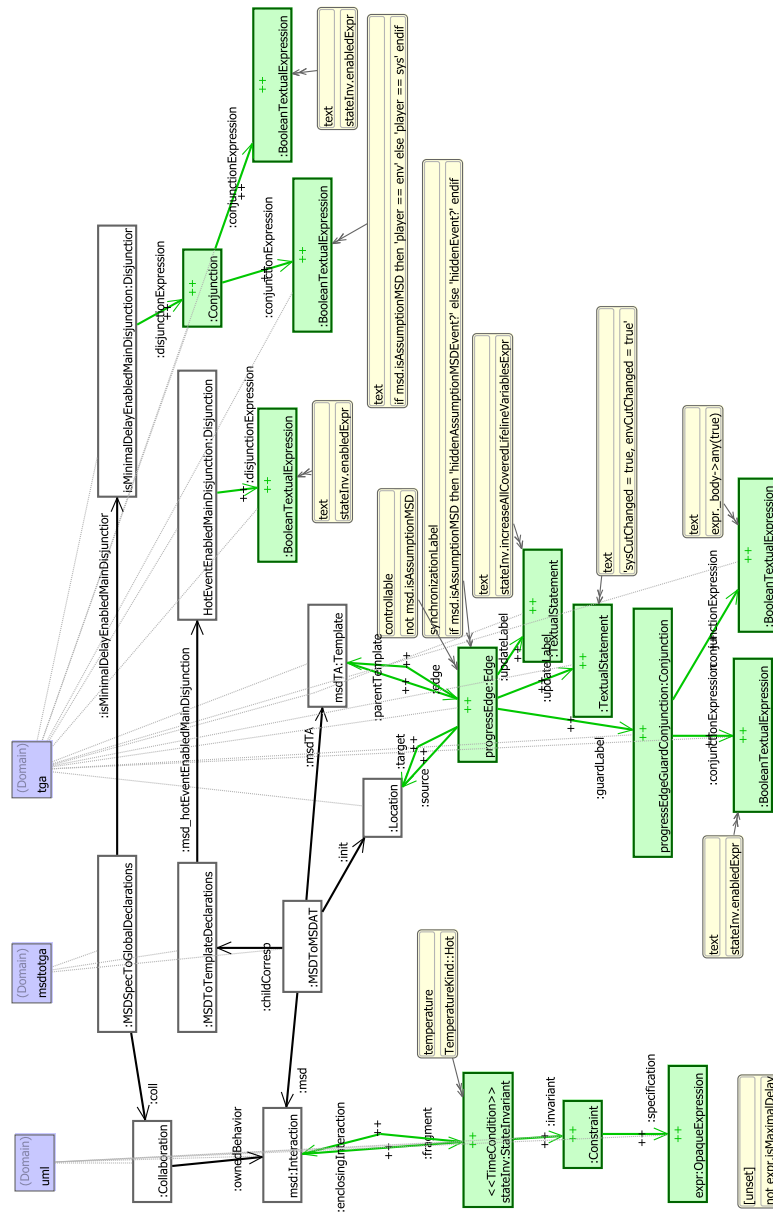


Figure B.37: The TGG rule HotMinimalDelay (mapping a hot minimal delay to an edge representing the progression of the delay in the corresponding MSD automaton template as well as parts in global functions)

Examples

This appendix chapter presents a number of example MSD specifications that were created to demonstrate, test and benchmark the the SCENARIOTOOLS simulation and synthesis features that were developed within the scope of this thesis.

Section C.1 describes a comprehensive RailCab specification that was modeled with the SCENARIOTOOLS editors and which can be simulated using the SCENARIOTOOLS implementation of the play-out algorithm.

Section C.2 presents one use case specification from the RailCab specification in detail that resembles a case where the simulation may run into an avoidable violation during the execution by naive play-out. Because the violation is an avoidable violation, a controller can be successfully synthesized from the use case specification that contains a strategy to always avoid the avoidable violation. If the simulation is executed guided by this controller as explained in Chap. 5, the simulation can always avoid the avoidable violation. Section C.2 in particular presents excerpts from the controller as it is generated by UPPAAL TIGA.

Section C.3 presents a timed example specification of an industrial production robot, called the *production cell*, for which, depending on the values selected for certain time intervals, a controller can be successfully synthesized by UPPAAL TIGA. In particular, the section additionally presents a variant of the production cell specification that was *decomposed* according to the compositional synthesis technique presented in Sect. 4.6. The section presents benchmark results of the times required for the TGG transformation and the synthesis by UPPAAL TIGA.

Last, Sect. C.4 presents some technical timed and untimed MSD specifications that are used to benchmark the efficiency of the synthesis.

C.1 Simulating an example RailCab specification

This section overviews a comprehensive RailCab example specification that can be *modeled and simulated* with SCENARIOTOOLS. The specification integrates multiple use case specifications that describe how the RailCab and switch- and track section controls interact when RailCabs move in a track system. The example also integrates a use case specification that describes how different *modules within the RailCab* interact for the *energy management* of the RailCab [ADG⁺09, Sect. 2.1.6, pp. 87].

Furthermore, the example contains a use case that illustrates how an *avoidable violation* may occur during the simulation of the specification. A controller that can be successfully synthesized from the use case specification can be combined with the simulation in SCENARIOTOOLS so that the avoidable violation is always avoided. The use case specification and parts of the controller resulting from the synthesis are shown in Sect. C.2. Before coming to this use case specification, an overview of the example specification is given in Sect. C.1.1. Section C.1.2 to C.1.4 explain some use case specifications in more detail.

The example specification consists of nine class diagrams, nine collaboration diagrams, and 27 MSDs. Therefore not all details of the example specification will be explained here. If the reader wishes to inspect the example in more detail or wishes to test the simulation, SCENARIOTOOLS and the example specification can be downloaded and installed by following the instructions on the SCENARIOTOOLS website:

<http://www.cs.upb.de/index.php?id=scenariotools>

C.1.1 Example specification overview

The RailCab example specification consists of eleven packages that are shown in Fig. C.1. On the top, there is the package RailCabBase, which defines the base class model of the RailCab system. Below the RailCabBase package, there are a number of packages that contain use case specifications, indicated by the stereotype «MSDSpecification». The «merge» relationships between these packages represent dependencies among the packages. The use case specification packages depend on the RailCabBase package or on other use case specification packages. The bottom of the diagram shows the package RailCabIntegrated, which merges all the use case specifications. From this package, an ECore package is created, based on which an instance system for the simulation is created as explained in Sect. 7.3.

The class model defined in the package RailCabBase is shown in Fig. C.2. This class model defines RailCab systems, which consists of an environment, zero to many RailCabs and zero to many track section controls. A track section control can have a next track section control and zero to many registered RailCabs. A RailCab can have one current track section control.

The use case specifications specify the following behavioral aspects of the RailCab system: First, the use case Drive onto track section describes how a RailCab registers at the control of the next track section when it approaches

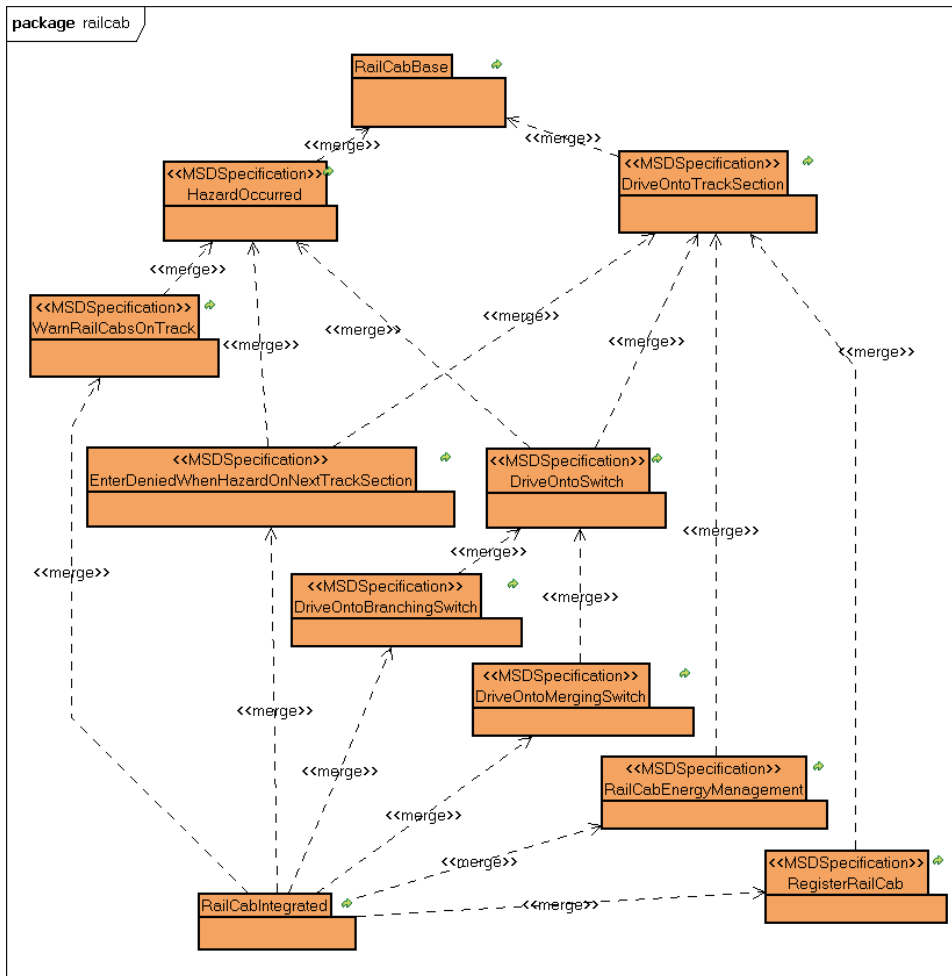


Figure C.1: A class diagram illustrating the merge relationships between the different use case specification packages

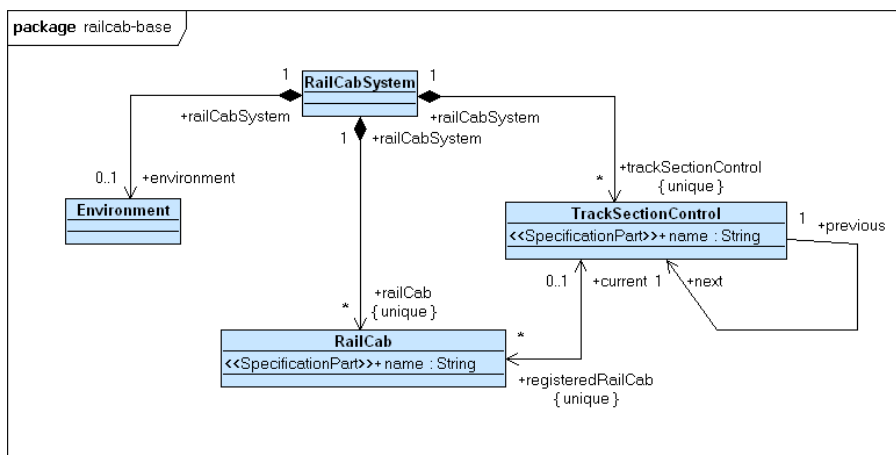


Figure C.2: The class model defined in the package RailCabBase

the end of its current track section. When a RailCab registers at the next track section control, the link to its current track section control is reset to the next track section control as described in the use case `Register RailCab`. The use case `Hazard occurred` describes that a RailCab must report a hazard to its current track section control when it detects an obstacle on the track. The use case `Warn RailCabs on track` describes that when a RailCab reports a hazard to its current track section control, it must also warn other RailCabs that are currently driving on the same track section. The use case `Enter denied when hazard on next track section` describes that a track section control must not allow a RailCab to enter when a hazard occurred on the track section. When a RailCab is not allowed to enter a next track section, it must stop, and will send a repeated request to enter the next track section when the hazard is resolved. The use cases `Drive onto switch`, `Drive onto branching switch`, and `Drive onto merging switch` introduce the classes `BranchingSwitchControl` and `MergingSwitchControl` and describe for example that a RailCab must not enter a merging switch if there is currently another RailCab registered at the merging switch control. These use cases also describe that a RailCab must not enter a switch if a hazard was reported on a track section immediately following the switch (as explained in the introduction, see Fig. 1.3). Next, the use case `RailCab Energy Management` introduces a number of modules that the RailCab consists of and that are involved in the RailCab's energy management. This use case specifies a protocol for how these modules coordinate on a specific driving profile prior to entering a track section.

C.1.2 Use case `DriveOntoTrackSection`

The use case `Drive onto track section` describes how a RailCab registers at the control of the next track section when it approaches the end of its current track section. As shown in the collaboration diagram in Fig. C.3¹, the use case involves the environment, a RailCab, its current and next track section controls, and the RailCab's route planner component. The MSDs in this use case specification are shown in Fig. C.4 to C.7.

The MSD `RequestEnterAtEndOfTrackSection` describes that the RailCab, upon approaching the end of the track section (`endOfTS`), must ask its route planner component for the track section control that the RailCab must register at next. The route planner replies with a reference to the next track section control as a parameter (`setNext(nextTSC)`). Within the MSD `RequestEnterAtEndOfTrackSection`, there is no value specified for the variable `nextTSC`, so the next track section control is not yet determined. Normally, the next track section control will be the track section control which is the next track section control of the RailCab's current track section control, i.e., referenced by the next reference (see the class diagram in Fig. C.2). It is described in the MSD `SetDefaultNextTSC` that the next track section control should be the next track section control of the RailCab's current track section (Fig. C.5). Usually, the `setNext` message

¹ In the collaboration diagrams shown here, the environment roles are not represented by a could symbol; this is currently not supported by the composite structure editor. The role `env:Environment` is always the environment role. There are no connectors shown between the roles in the collaboration diagrams, as it is not mandatory to include them.

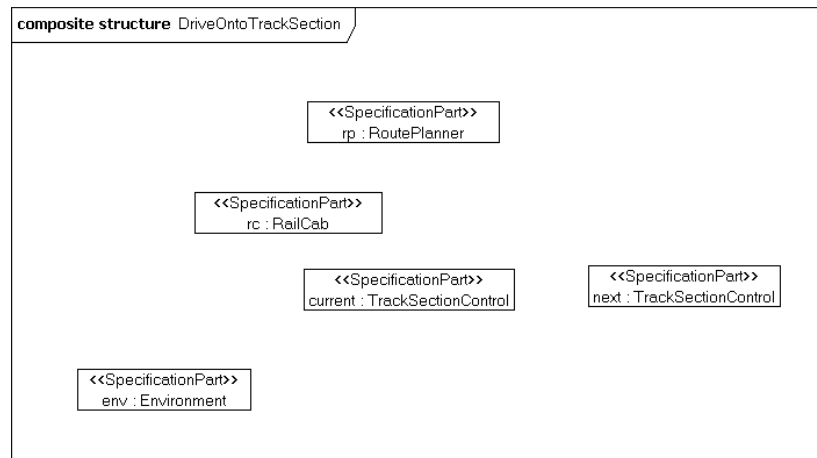


Figure C.3: The collaboration diagram in the Drive onto track section use case specification

in the MSD `SetDefaultNextTSC` will be unified with the `setNext` message in the MSD `RequestEnterAtEndOfTrackSection`, thereby binding the result of the OCL expression `rc.current.next` to the variable `nextTSC`. In the case where the `RailCab` is on a branching switch, as we will see shortly, the route planner may choose another track section control as the next track section control. The MSD `RequestEnterAtEndOfTrackSection` then specifies that the `RailCab` must request the permission to enter at the next track section. The `requestEnter` message is sent to the lifeline `next`, which is bound depending on the value of the parameter variable `nextTSC` of the preceding `setNext` message. The next track section control must then reply whether entering the next track section is allowed or not (`enterAllowed`). Again, the parameter value of the `enterAllowed` is not determined within this MSD. The MSD `DefaultEnterAllowed` (Fig. C.6) specifies that the next track section control should allow approaching `RailCabs` to enter. However, if for example a hazard occurred on the next track section control (use case `Hazard occurred`) or the `RailCab` plans to drive onto a merging switch where already another `RailCab` is registered (`Drive onto merging switch`), the respective track section or switch control must not allow the approaching `RailCab` to enter. If the value bound to the variable `isAllowed` is true, then the `RailCab` must register at the next track section control and it must then unregister from the current track section control. All the above messages must be sent before the `RailCab` passes the point of the last safe break and enters the next track section.

The MSD `StopWhenEnterDenied` (Fig. C.7) specifies that if the `RailCab` is not allowed to enter the next track section, it must stop. This is described by a `stop` message to the environment. (Here the environment is not the physical environment of the `RailCab`, but resembles a set of lower-level services that are or will have to be provided by the `RailCab`.)

The class diagram in Fig. C.8 captures the extensions that the use case specification `Drive onto track section` introduces to the structural model of the

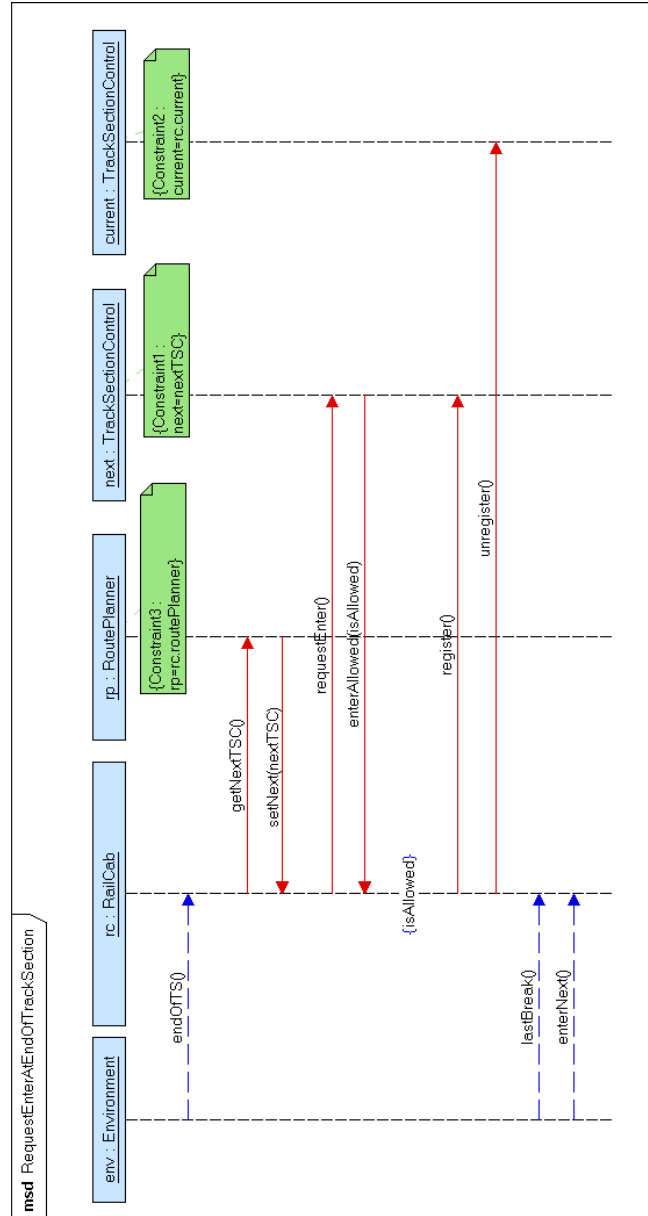


Figure C.4: The MSD RequestEnterAtEndOfTrackSection of the use case specification Drive onto track section

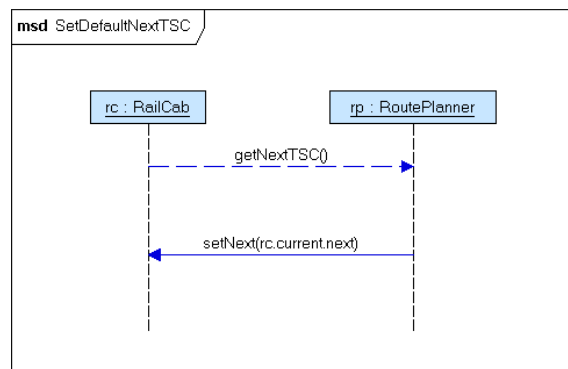


Figure C.5: The MSD SetDefaultNextTSC of the use case specification Drive onto track section

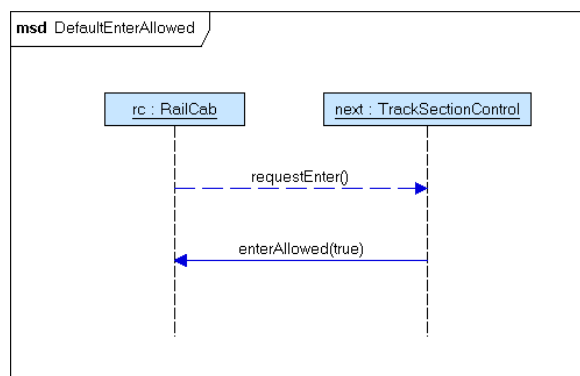


Figure C.6: The MSD DefaultEnterAllowed of the use case specification Drive onto track section

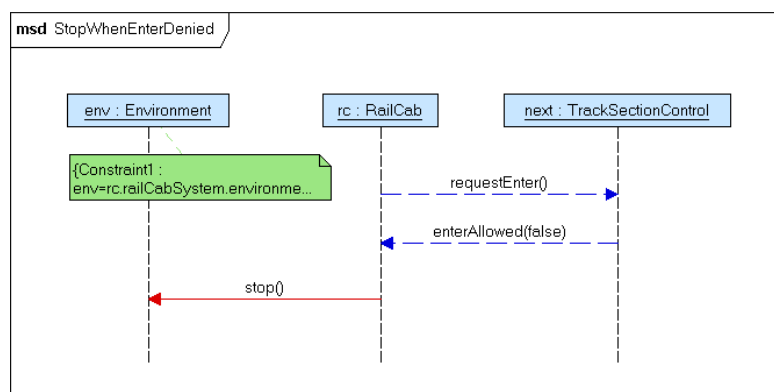


Figure C.7: The MSD StopWhenEnterDenied of the use case specification Drive onto track section

RailCab. It specifies that the RailCab has one route planner and may reference a next track section control which it is going to register with next (see above). Also, it defines operations for classes that specify which kinds of messages instances of these classes can receive. (Within SCENARIOTOOLS, the MSD editor automatically creates these operations when a new message kind is introduced while editing an MSD.)

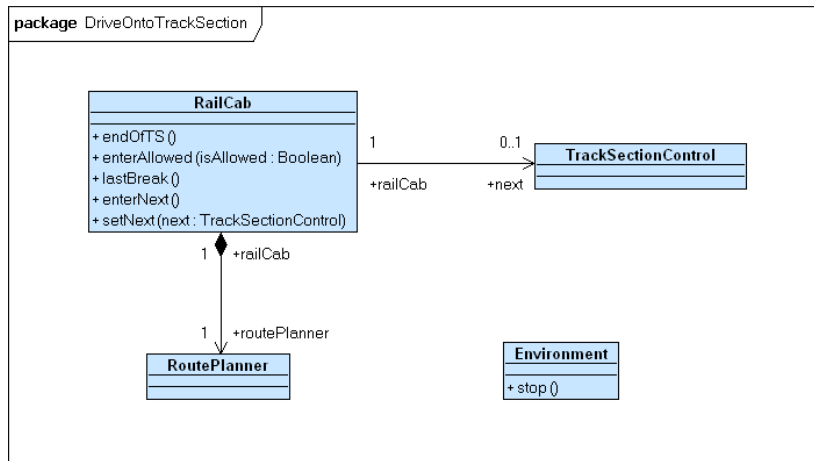


Figure C.8: The class diagram in the Drive onto track section use case specification

C.1.3 Use case DriveOntoBranchingSwitch

A look into the use case specification Drive onto branching switch gives some more interesting insights into the RailCab example specification. The use case again describes a situation where a RailCab approaches the end of its current track section, but it describes in particular a situation where the next track section is a branching switch. The branching switch is controlled by a branching switch control, a special kind of track section control.

The instances interacting in this use case are specified in the collaboration diagram shown in Fig. C.9. Here again the environment, a RailCab, and its route planner appear. Additionally, there is the branching switch control and the two subsequent track section controls. A branching switch connects a preceding track section with two subsequent track sections. One leaving straight, the other branching to the side. Accordingly, as shown in the class diagram of the use case specification (Fig. C.10), a branching switch control is a special kind of switch control that not only has a next track section control (see the base class model in Fig. C.2), but also has second next track section control that controls the track section following the branching exit of the switch. This track section control is referenced by the reference nextBranching.

If a RailCab approaches a branching switch, there are a number of cases where the RailCab must not be permitted to enter. One situation is the case where a hazard occurred on the branching switch. This is situation is covered in the use case Hazard occurred. Another case is the situation where a hazard

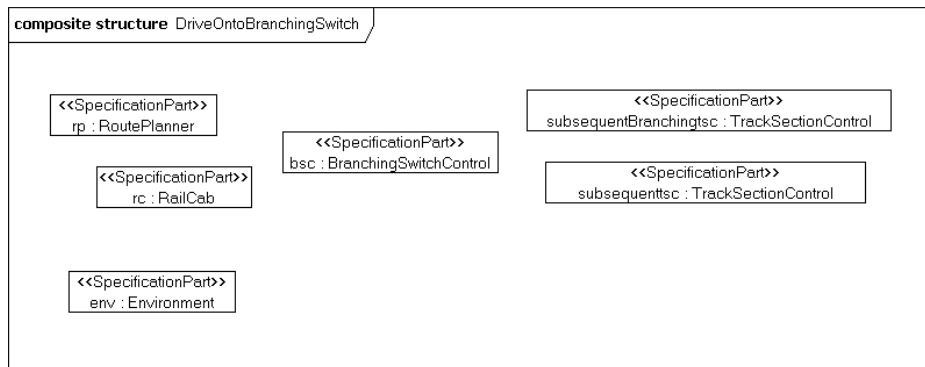


Figure C.9: The collaboration diagram in the Drive onto branching switch use case specification

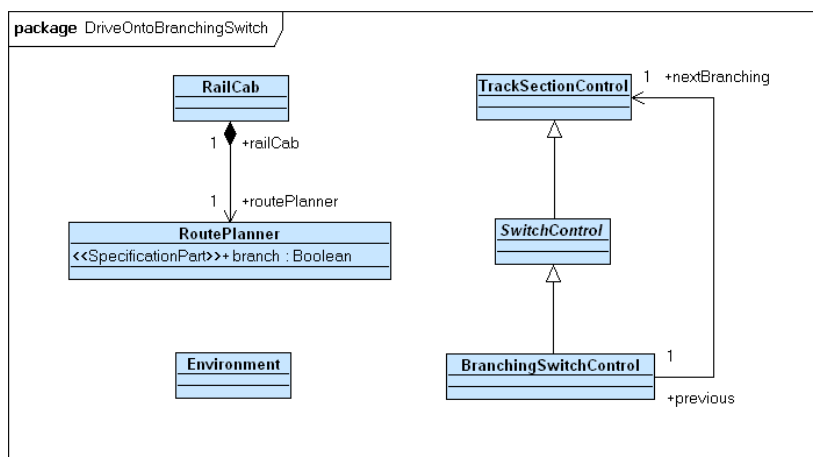


Figure C.10: The class diagram in the Drive onto branching switch use case specification

occurred on a subsequent track section. Due to the RailCab's doubly-fed linear drive technology, the RailCab will only have a limited breaking power on switches, which, in contrast to regular track sections, have no active magnetic field. It thus may happen that a RailCab will not always be able to break on a switch in order to avoid entering the next track section.

The MSD `EnterDeniedWhenHazardOnBranchingTrackSection` describes the scenario where a branching switch control receives a request to enter from an approaching RailCab, while there is a hazard on the track section following the branching exit of the switch. (The case where there is a hazard on the track section following the straight exit of the switch is described in the use case `Drive onto switch`.) The MSD specifies that when the RailCab sends a `requestEnter` message to a branching switch control, then the condition must be checked whether there is a hazard on the subsequent track section leaving the branching exit of the switch. This is specified by a cold condition that synchronizes the two lifelines `bsc` (representing the branching switch control) and `subsequent-`

Branchingtsc (representing the control of the subsequent track section leaving the branching exit of the switch). If this condition evaluates to true, then the branching switch control must not allow the RailCab to enter.

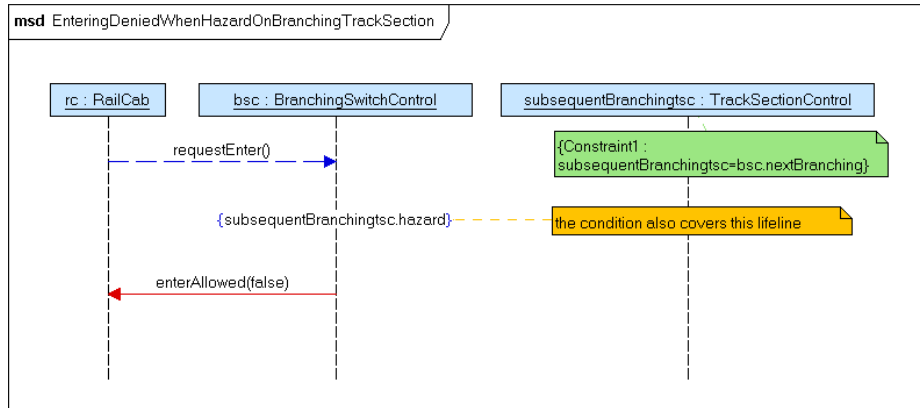


Figure C.11: The MSD EnterDeniedWhenHazardOnBranchingTrackSection of the use case specification Drive onto branching switch

As described above by the MSD StopWhenEnterDenied, if the next track section control does not allow the RailCab to enter the next track section, the RailCab will stop and wait. However, we want the RailCab to eventually start moving again. We could specify a certain time period in which the RailCab will again request the permission to enter. More simply, we specify that the RailCab is triggered to repeat its request to enter the next track section if the situation which caused the refusal of the request has changed. The MSD NotifyRailCab-WhenHazardOnSubsequentBranchingSwitchResolved describes that the RailCab should repeat its request to enter the next track section if a hazard that occurred on the track section leaving the branching exit of the branching switch (and that may have led to the refusal of the request to enter according to the MSD EnterDeniedWhenHazardOnBranchingTrackSection) is resolved. The MSD is triggered by a sequence of four events where, first, the RailCab requests the permission to enter a branching switch that is, second, refused by the branching switch control. Then, third, the hazard on the track section leaving the branching exit of the switch was resolved in the environment, fourth, leading to setting the hazard property of the respective track section control to false. (The use case Hazard occurred introduces the hazard property on track section controls to mark whether here is currently a hazard on a track section.) After this sequence of events, the branching switch control should notify the RailCab that the hazard was resolved. The RailCab should then in turn repeat its request to enter the next track section. The branching switch control should then allow the RailCab to enter. If there is for example another reason to refuse the request to enter, for example because there is a hazard remaining on the branching switch, the cold `enterAllowed(true)` message may be violated by another MSD where a message is enabled, requiring that the reply is `enterAllowed(false)` by a hot message.

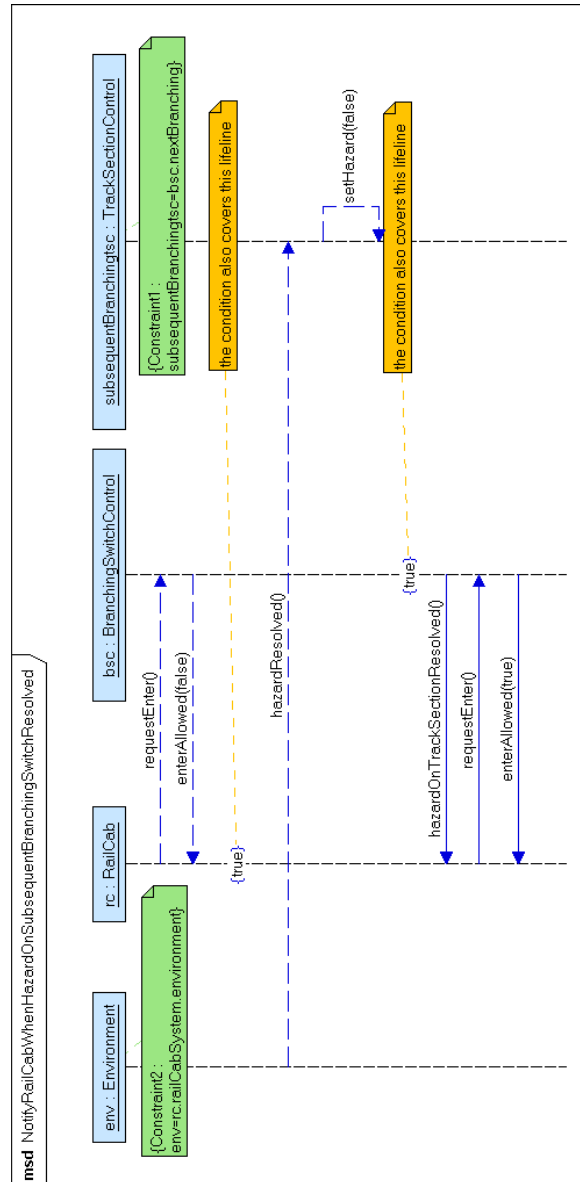


Figure C.12: The MSD NotifyRailCabWhenHazardOnSubsequentBranchingSwitchResolved of the use case specification Drive onto branching switch

The MSD `SetNextTrackSectionControlWhenBranchingOnSwitch` describes a situation where the RailCab is already on a branching switch and queries its route planner to determine the track section control that the RailCab shall register at next. Here the MSD determines the next track section control as follows. First, a cold condition asks whether the value of the property `branch` of the route planner is true and if the current track section control is indeed a branching switch control. The route planner has a property `branch` which encodes whether the current decision of the route planner is to branch or to continue moving straight. If the cold condition evaluates to false, the active copy of the MSD is discarded. Otherwise the subsequent assignment is executed, assigning the variable `nexttsc` with the reference to the control of the track section leaving the branching exit of the switch that the RailCab is currently on. The variable `nexttsc` is then bound and determines the parameter value of the message `setNext(nexttsc)` that the route planner sends to the RailCab in reply to the RailCab's query.

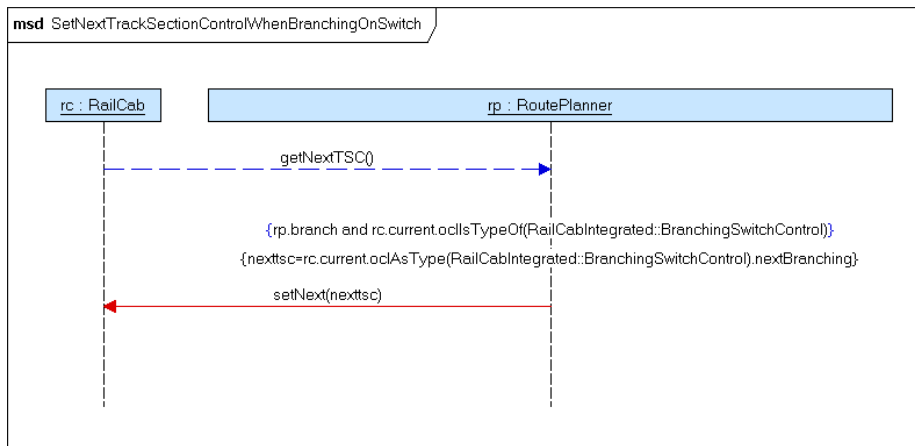


Figure C.13: The MSD `SetNextTrackSectionControlWhenBranchingOnSwitch` of the use case specification `Drive onto branching switch`

C.1.4 Use case `EnergyManagement`

The RailCab example specification furthermore contains a use case specification that describes how modules within the RailCab interact for the RailCabs energy management when the RailCab approaches the next track section. There are four RailCab modules involved in the RailCab energy management: the linear drive module, the energy supply module, the air gap adjustment system, and the active suspension and tilting module. The RailCab's energy management is currently under development at the University of Paderborn. In order to thoroughly test prototypes of the modules, first a distributed test bench will be created. Figure C.14 illustrates this test bench.

A conceptual design of the energy management was worked out and described in Adelt et al. [ADG⁺09, Sect. 2.1.6, pp. 87]. The conceptual design basically consist of an active structure diagram that describes the information

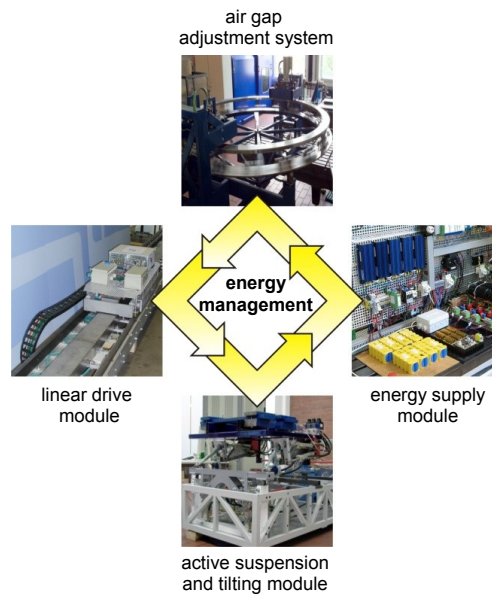


Figure C.14: The RailCab’s energy management requires the communication between different modules in the RailCab (AMS level) (taken from [ADG⁺09])

flows between the modules in the RailCab and an activity diagram that describes a sequence of events that should take place when a RailCab prepares to enter the next track section. In the following, this conceptual design is expressed by means of an MSD-based use case specification.

The collaboration diagram in Fig. C.15 shows the instances that interact to realize the RailCab’s energy management. These instances are the track section control, the RailCab, and the modules inside the RailCab: the drive module, air gap adjustment system, the suspension and tilting module, the energy supply module, and, last, the energy management, which is the central control component coordinating the energy management process. Figure C.16 shows the class model defined in the RailCab Energy Management use case specification that defines that the RailCab consists of the modules mentioned above.

The default energy management scenario is described by the MSD RailCab-EnergyManagementDefaultScenario shown in Fig. C.17. The scenario is triggered if a RailCab requests the permission to enter the next track section control and in reply the track section control grants the RailCab the permission to enter (`enterAllowed(true)`). If this happens, the MSD states that the track section control should also send a *profile of the track section* to the RailCab. The profile contains for example information about the maximum velocities allowed on different parts of the track section (*velocity profile*), information about the elevation of the different parts of the track section (*elevation profile*), and information about the power supply that is available on different parts of the track section (*power availability profile*). The message in this MSD abstracts from the concrete data that the track section control sends the RailCab. When designing this scenario in more detail, this data may be added as a parameter.

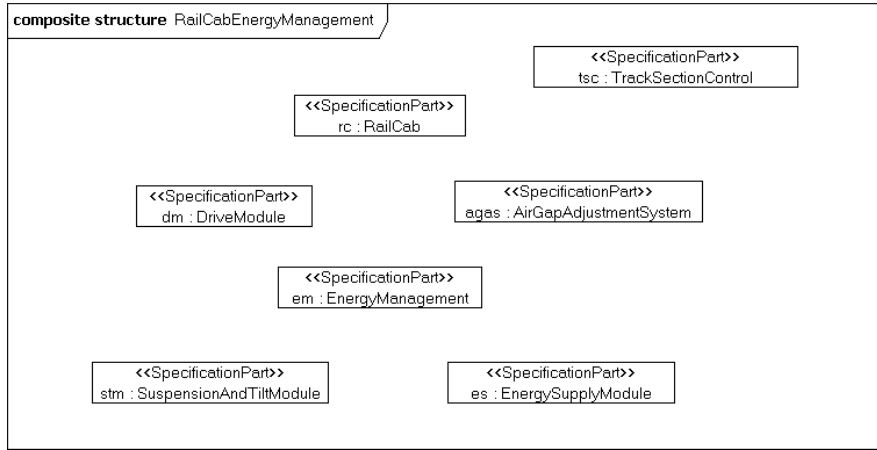


Figure C.15: The collaboration diagram in the RailCab Energy Management use case specification

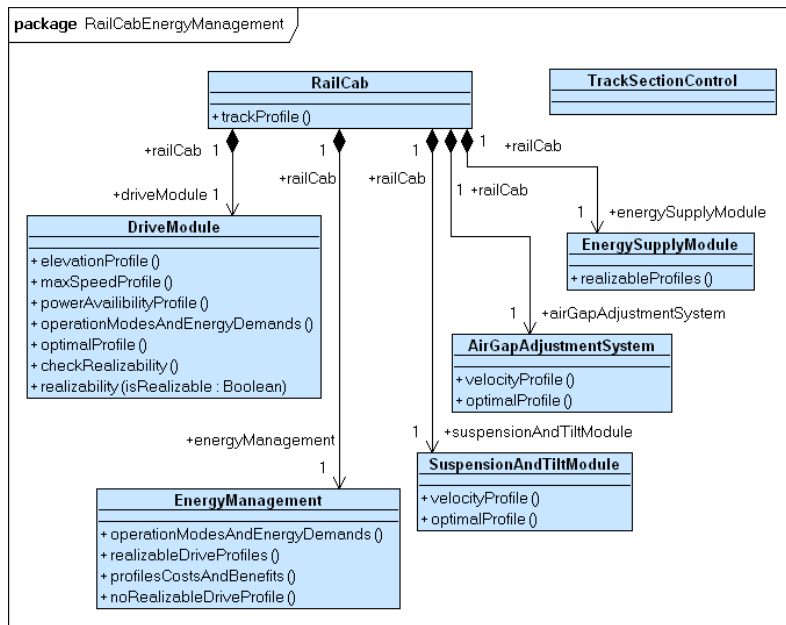


Figure C.16: The class diagram in the RailCab Energy Management use case specification

After receiving the track profile, the RailCab sends the elevation profile, the maximum speed profile and the power availability profile to the drive module. The drive module then calculates how fast it plans to drive on particular parts of the track section and sends this information, called the *velocity profile* to the suspension and tilting module as well as the air gap adjustment system. The higher the driving speed, the more energy the suspension and tilting module is likely to require for realizing a comfortable journey on the track section. Also, the higher the driving speed, the air gap adjustment system will require more energy to minimize the gap between the actor and stator field of the linear drive. The larger the gap becomes, the less energy efficient the drive will operate. Therefore, based on the velocity profile of the drive module, the suspension and tilting module and the air gap adjustment system calculate different possible operation modes and corresponding energy demands and send these different modes and demands to the energy management. The energy management collects this information and sends it to the drive module, which determines which of these suggested profiles are realizable. The energy demand by some of these profiles may exceed the energy that the drive module will be able to generate on the track section. The realizable profiles are then sent back to the energy management, which forwards these profiles to the energy supply module. The energy supply module consists of batteries and capacitors that can supply the RailCab with additional energy for a limited amount of time when the RailCab requires more energy than it can generate through the drive module, for example when driving uphill. The energy supply module calculates which of the realizable driving profiles will be the most cost efficient. Typically, the most cost efficient profiles are such where the energy supply must store the least energy or where the energy needs only be stored for short amounts of time. The energy supply sends the calculated costs and benefits of the different realizable driving profiles to the energy management which selects the optimal profile according to current optimization goals, like driving speed, driving comfort, or energy efficiency. The optimal driving profile is then sent to the drive module, the suspension and tilting module, and the air gap adjustment system, which then prepare to execute this profile when the RailCab finally enters the next track section.

All the messages in this scenario are cold, because it describes a sequence of events that should happen. But at this stage of the conceptual design, not all exceptions and alternatives in the energy management are anticipated. In the future, exceptions and alternative scenarios must be specified by additional MSDs. It may for example happen that the drive module cannot realize any of the driving profiles suggested by the other modules. This is anticipated in this MSD by including a cold forbidden message `noRealizableDriveProfile` at the bottom of the MSD. This indicates that a cold violation will occur and interrupt the described flow of event. What should or must be done in this case must be described by another MSD. However, the example does not yet specify these cases in more detail.

Next, Sect. C.1.5 overviews an instance model that is created based on the class models that are merged in the `RailCabIntegrated` package. The use case

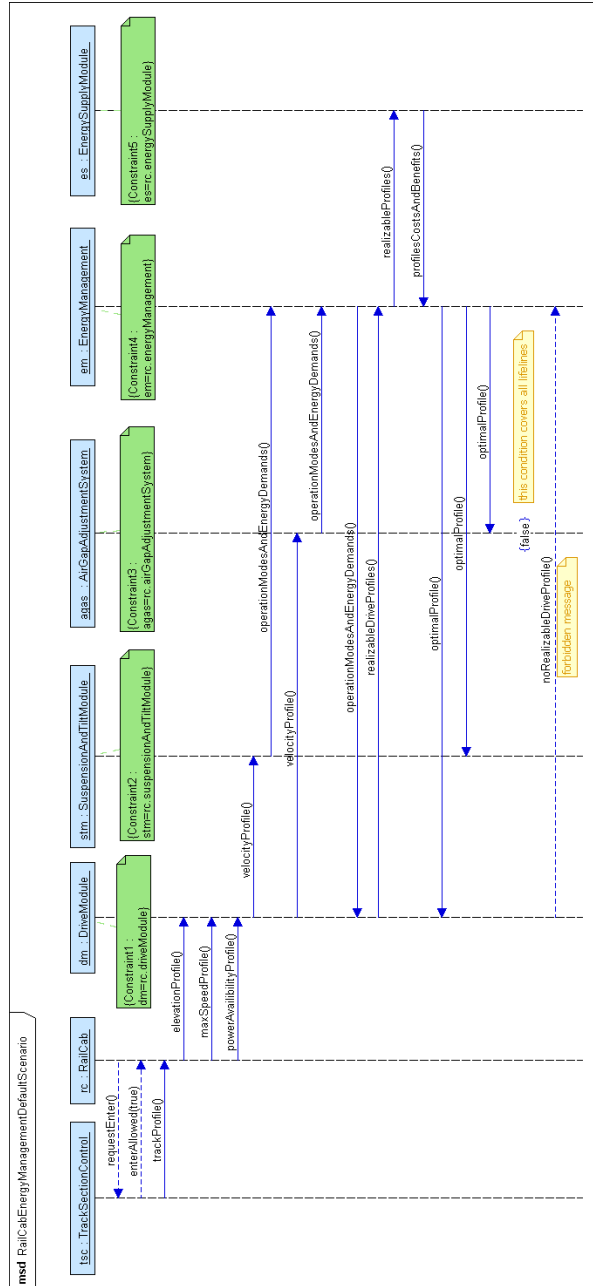


Figure C.17: The MSD SetNextTrackSectionControlWhenBranchingOnSwitch of the use case specification Drive onto branching switch

specification of the use case RailCab Obstacle Detected is presented in more detail in Sect. C.2.

C.1.5 The simulation model

As shown above in Fig. C.1, the package RailCabIntegrated integrates all the use case specifications in one package. By the UML-to-ECore transformation as explained in Sect. 7.3, an ECore model is created from this package and the merge relationships. The resulting ECore package is shown in Fig. C.18. This class model forms the basis for creating an instance model that can be simulated in SCENARIOTOOLS.

Figure C.19 shows a simple instance model that is created based on this ECore class model. It forms a simple circular track system with five track sections, two switches and two RailCabs driving in this system. The left side of the figure shows the instance system in the tree editor which EMF provides “for free” as a simple mechanism to create instance models. The right shows a more visually appealing sketch of the same system. In the future, it is desired to create editors for a better visualization of the instance models. This could be done, for example using the Graphical Modeling Framework (GMF)². But most importantly, it would be desirable to extend such editors so that the interactions of the instances are nicely visualized.

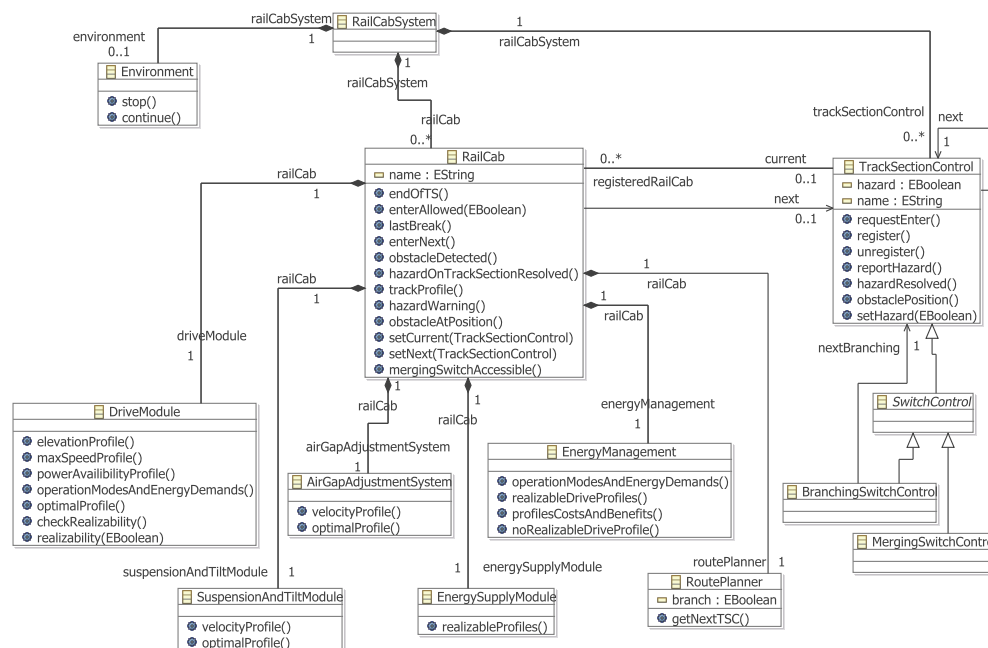


Figure C.18: The ECore class model resulting from the UML-to-Ecore transformation of the package RailCabIntegrated

²www.eclipse.org/gmf

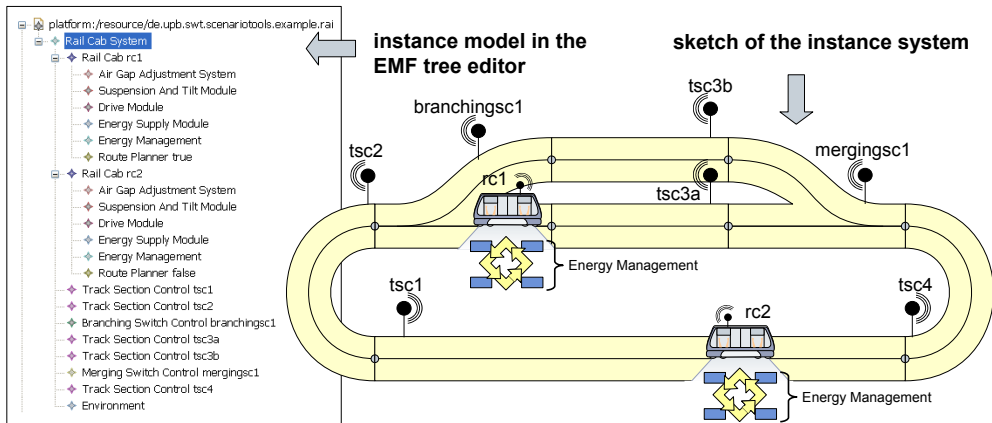


Figure C.19: An instance model of a simple track system with five track sections, two switches and two RailCabs

The user interface of SCENARIOTOOLS when simulating this example specification is shown in Fig. C.20. The main window shows the MSD RequestEnterAtEndOfTrackSection with its current cut. On the top right, there is the active MSDs view, which show the currently active MSDs. On the middle right, there is the trace view, showing a list of recorded events. At the bottom, there is the events view, showing the currently active events that the user can choose from to execute next.

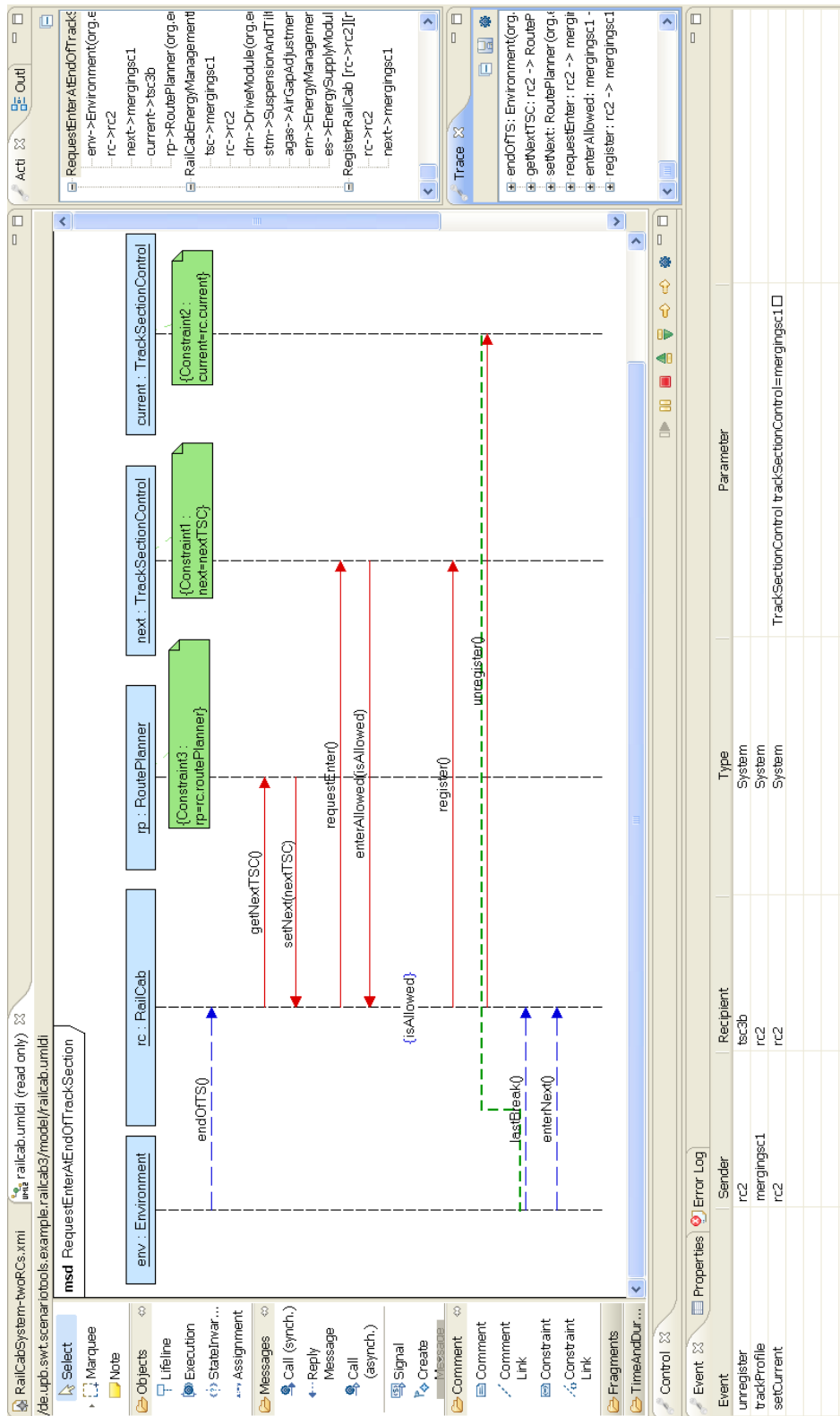


Figure C.20: A screenshot of the SCENARIOTOOLS use interface

C.2 The symbiosis of synthesis and simulation – the use case “Warn RailCabs On Track”

The use case Warn RailCabs on track is a simple example of an untimed use case where the simulation of the contained MSDs by the play-out algorithm may run into avoidable violations. (See also Sect. 5.2.) This section presents the specification of this use case in detail and describes the resulting controller that UPPAAL TIGA can successfully synthesize from the use case specification. This controller can be used by the SCENARIOTOOLS simulation in order to guide the play-out of the MSD specification presented in Sect. C.1.

C.2.1 Description of the use case “Warn RailCabs On Track”

Figure C.21 shows a sketch of the use case. The use case specifies that if a RailCab detects an obstacle on the track section, it must report to its current track section control that a hazard occurred on the track section. The RailCab must also send the position of the obstacle to the track section control. Then the track section control must warn other RailCabs on the track section and inform them about the position of an obstacle on the track section.

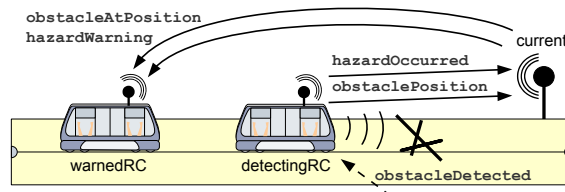


Figure C.21: A sketch of the situation described by the RailCab Obstacle Detected use case

C.2.2 The specification of the use case

The specification for the use case Warn RailCabs on track consists of the collaboration diagram and the MSDs that are shown in Fig. C.22 to C.25. (Note that this is a deliberately odd formalization of the use case in order to provide a small example where an avoidable violation may occur during play-out.) Some lifelines of the MSDs are equipped with binding expression. These are required for the dynamic binding of the lifelines during the simulation. For the synthesis, however, these binding expressions are ignored, since the synthesis assumes that the lifelines are static and the roles in the collaboration diagram represent the static object system.

The collaboration shows that the use case specifies the behavior of four roles, representing the environment, the track section control and two RailCabs. One role represents the RailCab that detects the obstacle; the other represents the RailCab that is warned by the track section control.

Figure C.23 shows the MSD WarningWhenObstacleDetected. It describes that, if a RailCab detects an obstacle (`obstacleDetected`), it must report a

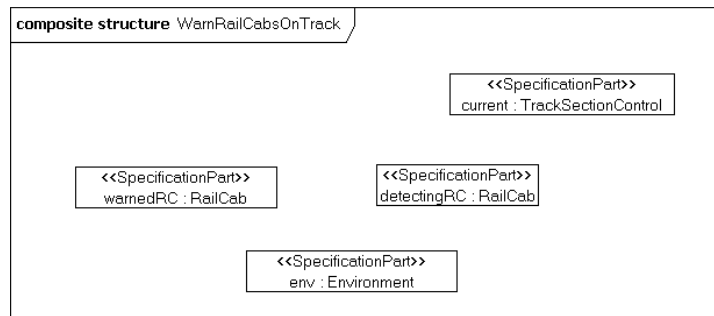


Figure C.22: The collaboration diagram of the Warn RailCabs on track use case specification

hazard to its current track section control (**reportHazard**). The track section control must then send a warning (**hazardWarning**) and the obstacle position (**obstacleAtPosition**) to the other RailCabs driving on the the same track section.

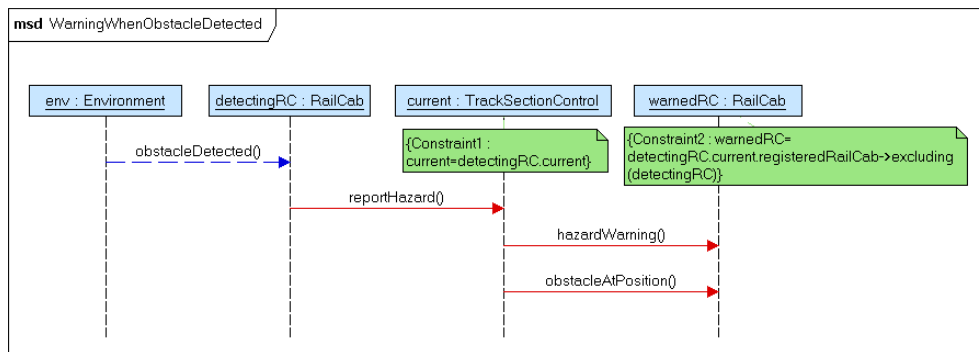


Figure C.23: The MSD WarningWhenObstacleDetected

Figure C.24 shows the MSD ReportObstaclePosition. It states that if a Rail-Cab detects an obstacle, it must send the position of the obstacle to its current track section control (**obstaclePosition**), so that the track section control then sends this information to the warned RailCab (**obstacleAtPosition**).

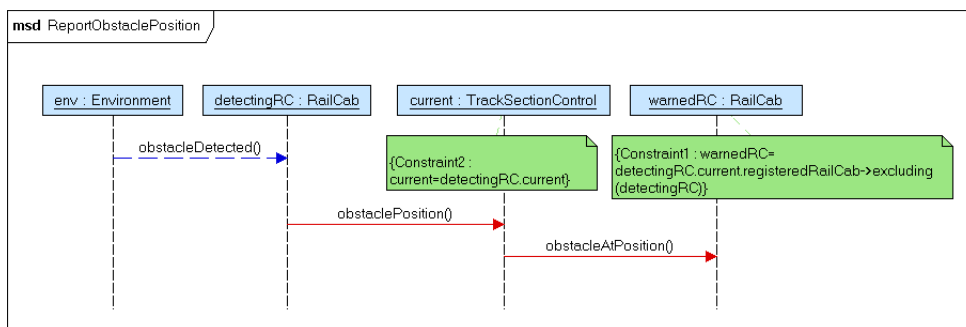


Figure C.24: The MSD ReportObstaclePosition

The MSD `ReportObstaclePositionAndIssueWarning` is shown in Fig. C.25. The MSD states that if a `RailCab` reports a hazard to its current track section control and also sends the position of an obstacle, the track section control must send the position of the obstacle and the hazard warning to the warned `RailCab`.

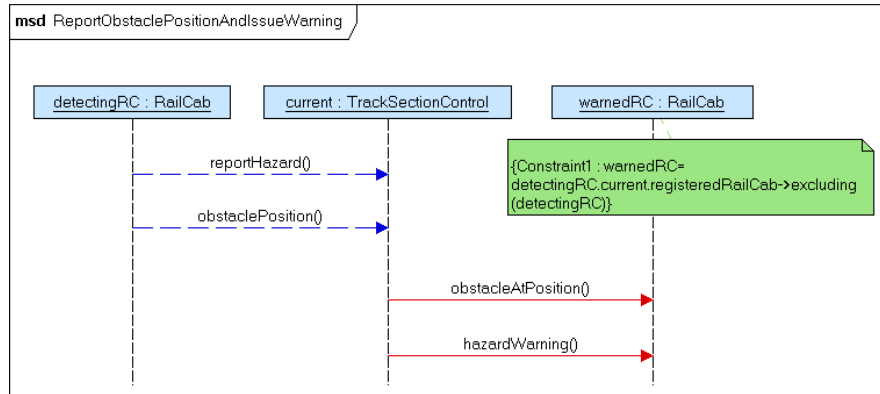


Figure C.25: The MSD `ReportObstaclePositionAndIssueWarning`

C.2.3 Avoidable violating runs

The problem with this specification is that if a `RailCab` detects an obstacle (`obstacleDetected`) and then sends the message `reportHazard` before sending the message `obstaclePosition`, there will be a hot violation. This is because the active copy of `WarningWhenObstacleDetected` requires `hazardWarning` to be sent followed by `obstacleAtPosition` and the active copy of `ReportObstaclePositionAndIssueWarning` requires a different order of events, namely `obstacleAtPosition` to be sent followed by `hazardWarning`.

This hot violation can however be avoided by sending (1) `obstaclePosition` before (2) `reportHazard`. This would avoid the activation of `ReportObstaclePositionAndIssueWarning`, and (3) `hazardWarning` followed by (4) `obstacleAtPosition` could be sent. Another admissible reaction to `obstacleDetected` is to send (1) `reportHazard`, (2) `hazardWarning`, (3) `obstaclePosition`, and last (4) `obstacleAtPosition`.

C.2.4 The controller synthesized from the use case specification

The synthesis technique presented in Chap. 4 allows us to automatically find such admissible runs. Listing C.1 shows an excerpt of the winning strategy that can be synthesized by UPPAAL TIGA from the corresponding TGA system and the *AGAF* winning condition as discussed in Sect. 4.4 (Prop. 4.1).

The synthesis settings

Because we want to use this strategy to guide the play-out of the specification, we have to employ synthesis in the following way.

- we have to synthesize a *complete* strategy, which means that we have to calculate all the winning actions for all winning states as explained in Sect. 5.2.2. (This is done by using the `verifytga` command of UPPAAL TIGA with the `-w2` option.)
- we have to synthesize a strategy where system messages may not only be sent when they are active. A strategy where the system can only execute messages when they are active on one of the MSDs is not sufficient: if we simulate an MSD specification where the use case Warn RailCabs on track is combined with other use case specifications, it may be that there are other active MSDs during the simulation where the messages appearing in the use case Warn RailCabs on track are active, even though they are not active in any MSD of Warn RailCabs on track (see the discussion on *consistency* vs. *consistent executability* Sect. 4.7.2).

The kinds of states in the synthesized controller

Listing C.1 shows part of the controller that is generated by UPPAAL TIGA. Such a controller consists of a list of states in the TGA system and directives to either wait for the next environment event to occur, or to take a particular transition from these states. From all the states in the generated controller, only a subset is of interest for a system, namely those where the system automaton (its instance is named *systemProcess*) is in the location `systemActive`. In this location, the controller either tells the system to wait or that it must take a particular step, see the explanation in Sect. 5.2.2.

A closer look at some states in the synthesized controller

Let us now take a closer look at the states displayed in Listing C.1:

The first state that is shown in the listing is the state where each MSD is in the initial cut. Here, the system can either choose to wait, or it may send any of the system messages appearing in the specification. The controller allows the system to do so even though these messages are not active in this state (i.e., there is no executed message enabled which represents this event). If we just synthesize a controller for the *consistent executability*, the resulting controller would not allow for these messages to be sent by the system.

The second state resembles the super-cut that is reached after `obstacleDetected` has occurred. Here the controller prescribes two alternative transitions, to either send the message `reportHazard`, or to send the message `obstaclePosition`. The controller does not allow the system to send the messages `obstacleAtPosition` or `hazardWarning`, nor does it allow the system to do nothing.

The third state listed below corresponds to the super-cut that is reached after the system chose to send the message `obstaclePosition`. Here the directive is to next send the message `reportHazard`. The controller does not allow the system to send the messages `obstaclePosition`, `obstacleAtPosition`, or `hazardWarning`, nor does it allow the system to do nothing.

The fourth state shown in the listing resembles the super-cut that is reached if the system chose to send `reportHazard` after `obstacleDetected` occurred.

Then the directive is to next send the message `hazardWarning`. The controller does not allow the system to send the messages `obstaclePosition`, `obstacleAtPosition`, or `reportHazard`, nor does it allow the system to do nothing.

The controller of course contains more states. In sum, the controller strategy contains 120 states. The remaining states will not be explained further. The strategy can be inspected in more detail after installing SCENARIOTOOLS and the RailCab MSD specification example.

Listing C.1: Excerpt from the controller synthesized from the specification of the use case Warn RailCabs on track

```

Strategy to win:
...

/*
 * All MSDs in the initial cut -- note that in this
 * state no message is active (i.e., no executed
 * message is enabled). Nevertheless, this controller
 * allows for any system message appearing in the
 * use case specification to occur:
 * 1. "obstacleAtPosition"
 * 2. "reportHazard"
 * 3. "obstaclePosition"
 * 4. "hazardWarning"
 * (see transitions 2-5)
 * Alternatively, the system can decide to do nothing
 * (return to 'systemProcess.systemInactive', first
 * transition).
 */
State: (
systemProcess.systemActive
environmentProcess.environmentInitial
WarningWhenObstacleDetected_inst.initial
ReportObstaclePosition_inst.initial
ReportObstaclePositionAndIssueWarning_inst.initial )
event=0 hotViolation=0 hotEnvViolation=0
WarningWhenObstacleDetected_env=0
WarningWhenObstacleDetected_detectingRC=0
WarningWhenObstacleDetected_current=0
WarningWhenObstacleDetected_warnedRC=0
ReportObstaclePosition_env=0
ReportObstaclePosition_detectingRC=0
ReportObstaclePosition_current=0
ReportObstaclePosition_warnedRC=0
ReportObstaclePositionAndIssueWarning_detectingRC=0
ReportObstaclePositionAndIssueWarning_current=0
ReportObstaclePositionAndIssueWarning_warnedRC=0
systemProcess.block=1
sysCutChanged=1 envCutChanged=1
When you are in true, take transition
  systemProcess.systemActive->systemProcess.systemInactive
  { 1, tau, 1 }
When you are in true, take transition
  systemProcess.systemActive->systemProcess.produceEvent
  { 1, tau, event := current_warnedRC_obstacleAtPosition }
When you are in true, take transition
  systemProcess.systemActive->systemProcess.produceEvent
  { 1, tau, event := detectingRC_current_reportHazard }
When you are in true, take transition
  systemProcess.systemActive->systemProcess.produceEvent
  { 1, tau, event := detectingRC_current_obstaclePosition }
When you are in true, take transition
  systemProcess.systemActive->systemProcess.produceEvent
  { 1, tau, event := current_warnedRC_hazardWarning }

...

/*
 * State after "obstacleDetected" -- the system must produce
 * either "reportHazard" or "obstaclePosition".
 * The system must not produce "obstacleAtPosition" or

```

```

    * "hazardWarning" to not violate the specification.
    */
    State: (
    systemProcess.systemActive
    environmentProcess.environmentInitial
    WarningWhenObstacleDetected_inst.initial
    ReportObstaclePosition_inst.initial
    ReportObstaclePositionAndIssueWarning_inst.initial )
    event=2 hotViolation=0 hotEnvViolation=0
    WarningWhenObstacleDetected_env=1
    WarningWhenObstacleDetected_detectingRC=1
    WarningWhenObstacleDetected_current=1
    WarningWhenObstacleDetected_warnedRC=1
    ReportObstaclePosition_env=1
    ReportObstaclePosition_detectingRC=1
    ReportObstaclePosition_current=1
    ReportObstaclePosition_warnedRC=1
    ReportObstaclePositionAndIssueWarning_detectingRC=0
    ReportObstaclePositionAndIssueWarning_current=0
    ReportObstaclePositionAndIssueWarning_warnedRC=0
    systemProcess.block=1
    sysCutChanged=1 envCutChanged=1
    When you are in true, take transition
        systemProcess.systemActive->systemProcess.produceEvent
        { 1, tau, event := detectingRC_current_reportHazard }
    When you are in true, take transition
        systemProcess.systemActive->systemProcess.produceEvent
        { 1, tau, event := detectingRC_current_obstaclePosition }

    ...

    /*
    * State after "obstacleDetected", "obstaclePosition" --
    * the system must produce "reportHazard".
    * The system must not produce "obstacleAtPosition",
    * "obstaclePosition" or "hazardWarning" to not
    * violate the specification.
    */
    State: (
    systemProcess.systemActive
    environmentProcess.environmentInitial
    WarningWhenObstacleDetected_inst.initial
    ReportObstaclePosition_inst.initial
    ReportObstaclePositionAndIssueWarning_inst.initial )
    event=4 hotViolation=0 hotEnvViolation=0
    WarningWhenObstacleDetected_env=1
    WarningWhenObstacleDetected_detectingRC=1
    WarningWhenObstacleDetected_current=1
    WarningWhenObstacleDetected_warnedRC=1
    ReportObstaclePosition_env=1
    ReportObstaclePosition_detectingRC=2
    ReportObstaclePosition_current=2
    ReportObstaclePosition_warnedRC=1
    ReportObstaclePositionAndIssueWarning_detectingRC=0
    ReportObstaclePositionAndIssueWarning_current=0
    ReportObstaclePositionAndIssueWarning_warnedRC=0
    systemProcess.block=1
    sysCutChanged=1 envCutChanged=1
    When you are in true, take transition
        systemProcess.systemActive->systemProcess.produceEvent
        { 1, tau, event := detectingRC_current_reportHazard }

    ...

    /*
    * State after "obstacleDetected", "hazardOccurred" --
    * the system must produce "hazardWarning".
    * The system must not produce "obstacleAtPosition",
    * "obstaclePosition" or "reportHazard" to not
    * violate the specification.
    */
    State: (
    systemProcess.systemActive
    environmentProcess.environmentInitial
    WarningWhenObstacleDetected_inst.initial
    ReportObstaclePosition_inst.initial
    ReportObstaclePositionAndIssueWarning_inst.initial )
    event=3 hotViolation=0 hotEnvViolation=0
    WarningWhenObstacleDetected_env=1

```

```
WarningWhenObstacleDetected_detectingRC=2
WarningWhenObstacleDetected_current=2
WarningWhenObstacleDetected_warnedRC=1
ReportObstaclePosition_env=1
ReportObstaclePosition_detectingRC=2
ReportObstaclePosition_current=2
ReportObstaclePosition_warnedRC=1
ReportObstaclePositionAndIssueWarning_detectingRC=1
ReportObstaclePositionAndIssueWarning_current=1
ReportObstaclePositionAndIssueWarning_warnedRC=1
systemProcess.block=0
sysCutChanged=1 envCutChanged=1
When you are in true, take transition
  systemProcess.systemActive->systemProcess.produceEvent
    { 1, tau, event := current_warnedRC_hazardWarning }
...

```

C.3 Synthesis example – the production cell

The example presented in this section is that of an industrial production robot, called the *production cell*, which processes metal blanks into plates. This example was presented as a case study for a safety-critical real-time reactive system within the KORSO project [LL94, LL06]. (KORSO is an abbreviation for the German title “Korrekte Software”, “correct software” [BJ95].) The original production cell is a system consisting of 14 sensors and 13 actuators [LL06]. In the following, a simplified version of the example is considered. A similar example was also used to benchmark UPPAAL TIGA by Cassez et al. [CDF⁺05].

The purpose of this example is, first, to show how to *model the specification of a more comprehensive, practical example system* with timed MSDs. This example especially demonstrates that in order to realize the requirements of the production cell, *it is of the essence to make assumptions about the behavior of the system’s environment*, in particular its mechanical parts, e.g., the movement of robot arms. Furthermore, this specification serves as an example to *demonstrate the compositional synthesis technique* that was described in Sect. 4.6. Last, the purpose of this example is to *validate the synthesis technique*. The main results are

- The synthesis produces *correct results* for this example, i.e., the synthesis reports the specification to be consistently executable for minimal and maximal delays that were anticipated to be realizable by the production cell. Furthermore, the synthesis reports the specification not to be consistently executable for minimal and maximal delays that should not be realizable by the production cell.
- The example specification is *already too complex for checking its consistent executability using the AGAF winning condition* (see Sect. 4.4). UPPAAL TIGA runs out of memory (4 GB) while exploring the state space induced by the specification.
- Changing the synthesis setting such that the system *is not allowed to delay the execution of active events* can *drastically reduce the synthesis time*.
- *Decomposing the synthesis problem greatly reduces the complexity of the synthesis problem* and again *greatly reduces the overall synthesis time*.

In the following, Sect. C.3.1 describes the production example informally. Section C.3.2 then explains the MSD specification and presents results from synthesizing a controller from the specification with different values for certain minimal and maximal delays in the example. Section C.3.3 then introduces the decomposition of the production cell specification into two part specifications and it presents the synthesis times for the part specifications and different minimal and maximal delays.

C.3.1 Description of the production cell example

A sketch of the production cell is shown in at the bottom of Tab. C.1. It consists essentially of two transport belts, a press, and two robot arms. The production cell must work as follows: Metal blanks are transported to the production cell on the feed belt within certain time intervals. At the end of the feed belt, these

blanks arrive on a table and are detected by a sensor. If a blank is detected by the table sensor, it must be picked up by the first robot arm. This robot arm must then transport the blank to the press, where it must then be pressed to a plate.

Next, a second robot arm must pick up the plate and transport it to the deposit belt. It is required that a blank is picked up from the table before the next blank arrives. Also, the second robot arm must pick up the pressed plate before the first robot arm can place a new blank into the press. The controller of the production cell can control the press and the movement of the robot arms. The arrival of the blanks, the time for pressing the blanks to plates, and the time that the robot arms require for moving between the table and the press resp. the press and the deposit belt are uncontrollable, but certain minimal and maximal delays can be assumed. (Different concrete values for these delays will be chosen later on to evaluate the synthesis.)

The use case for this production cell is described in detail in Tab. C.1. Here just one use case for the production cell is considered, namely the processing of the arriving blanks. There could be further use cases, for example for exception handling, but these are not considered here. An MSD specification capturing these requirements is shown in Sect. C.3.2. For better readability, the paragraphs resp. sentences in the use case description are numbered. The MSDs representing these textual requirements in Sect. C.3.2 are numbered accordingly.

C.3.2 The MSD specification of the production cell

The collaboration diagram that represents the object system of the production cell and the MSDs representing the requirements of the use case are shown in Fig. C.26. The object system shows that the only controllable system component is the controller c:Controller. The other components of the production cell, the arms a:ArmA and b:ArmB, the press p:Press, and the table sensor ts:TableSensor are environment objects.

The requirement MSDs

The MSDs (1)-(4) formalize the behavior described in the use case in a quite straightforward way. The MSD ArmATransportBlankToPress (1) is triggered when a blank has arrived at the table. It describes that the blank must be picked up by arm a:ArmA, which then has to move to the press. Then arm a:ArmA must arrive at the press. Note that this is a hot environment message, which means that nothing must abort the sequence of events at this point. This event is, however, not controllable by the system; it is shown shortly that there are MSDs formulating environment assumptions that guarantee that arm a:ArmA, when told to move to the press, will arrive at the press (see ArmA-MoveFromTableToPressTimeAssumption (10) in Fig. C.27). When arm a:ArmA has arrived at the press, it must release the blank into the press and then move back to the table. Then it must eventually arrive at the table. As before, this is modeled by a hot message. Including this message in the MSD in particular expresses that the arm has to be back at the table before the next blank arrives.

Table C.1: Use Case Process Plate

Use Case: Process Plate	Nr. 1
<p>Requirements:</p> <p>When a blank plate arrives at the table, it must be picked up by arm A. Then the arm A has to move to the press and, when it has arrived, it must release the blank into the press. Arm A then has to move back to the table. Arm A must have arrived at the table before the next blank arrives (1).</p> <p>When arm A has released the blank into the press, the press must press the plate. When the pressing process is finished, then arm B must pick up the plate (2). When arm B has picked up the processed plate, it must transport the plate to the deposit belt (3). When arm B has arrived at the deposit belt, it must release the processed plate. After releasing the processed plate, arm B must move back to the press (4).</p> <p>Plates must only be released into the press by arm A if arm B has picked up the processed plate from the press (5).</p> <p>Arm A must not try to pick up the next blank before it has returned to the table (6). Arm B must not try to pick up the next pressed plate before it has returned to the press (7).</p>	
<p>Environment assumptions:</p> <p>In the initial configuration, there is no blank on the table, arm A is located at the table, and arm B is located at the press.</p> <p>The following minimal and maximal delays can be guaranteed by the environment:</p> <ol style="list-style-type: none"> Plates arrive on the table at the end of the feed belt in intervals of f seconds, where $f > FMIN$ (9). Arm A can move from the table to the press or from the press to the table in a seconds, where $AMIN \leq a \leq AMAX$ (10). Arm B can move from the press to the deposit belt or from the deposit belt to the press in b seconds, where $BMIN \leq b \leq BMAX$ (11). The press needs p seconds to process a blank plate, where $PMIN \leq p \leq PMAX$ (12). 	
<p>Sketch:</p>	

The MSD `PressPlateAfterArmAReleasesBlankPlate` (2) says that, after the blank was released by arm `a:ArmA`, the press `p` must press the blank to a plate and, when the pressing process is finished, arm `b:ArmB` must pick up the pressed plate. The MSD `ArmBTransportToDepositBeltAfterPickUpFromPress` (3) just states that arm `b:ArmB` must move to the deposit belt after picking up the plate from the press. MSD `ArmBReleasePlateAndReturnToPress` (4) states that arm `b:ArmB` must release the plate and return to the press after having arrived at the deposit belt. Note that the MSDs `ArmBTransportToDepositBeltAfterPickUpFromPress` (3) and `ArmBReleasePlateAndReturnToPress` (4) are not triggered by the environment events `arrivedAtDepositBelt` and `pressingFinished` as described by the textual requirements. Instead, these MSDs are triggered by the preceding system messages `moveToDepositBelt` and `press`. This is because otherwise these MSDs could be triggered at any time by the environment, thus easily leading to violations. The environment assumptions, which will be explained shortly, do not always restrict `arrivedAtDepositBelt` and `pressingFinished` from occurring.

The MSD `ArmAMustNotReleaseBlankBeforePlateRemovedByArmB` (5) specifies that `a:ArmA` must not release a blank into the press unless the previous pressed plate was picked up by `b:ArmB`. This is modeled by a hot forbidden message. The MSD states that it is not admissible to release a blank into the press twice unless the processed plate was picked up in between. Note that the MSD-to-TGA mapping requires that forbidden messages appear at the end of the MSD, separated by a cold `false` condition that marks the actual end of the MSD, see Sect. 4.3.4. Similarly, the MSDs `ArmAMustNotPickUpBlankBeforeReturnedToTable` (6) and `ArmBMustNotPickUpPlateBeforeReturnedToPress` (7) prohibit that blanks/plates are picked up by the robot arms `a:ArmA` resp. `b:ArmB` before they arrived at the according location.

The assumption MSDs

The MSD `BlankArrivalDelay` (9) describes that blanks may only arrive at the table within a certain minimal delay. The MSDs `ArmAMoveFromPressToTableTimeAssumption` (10) and `ArmAMoveFromTableToPressTimeAssumption` (10) specify that the robot arm `a:ArmA`, if told to move to the press resp. table, it will arrive there within a certain time interval. The hot forbidden messages in these MSDs state that if the arm is told to move to a desired location it will not end up at the opposite direction. Furthermore, if the system orders the robot arm to move to a location, but then orders it to return to its starting location, it is assumed that the robot arm reverses before reaching its original destination. For example, if the system orders the arm to move to the press, but then orders it to move back to the table before the arm has arrived at the press, the arm will return to the table and it will not reach the press. This behavior is modeled by a cold forbidden message at the bottom of the MSDs. Remember that an occurrence of an event that is represented by a cold forbidden message results in the active MSD being safely exited, even if the active MSD is in a hot cut. This behavior is not actually described in the use case text, but it

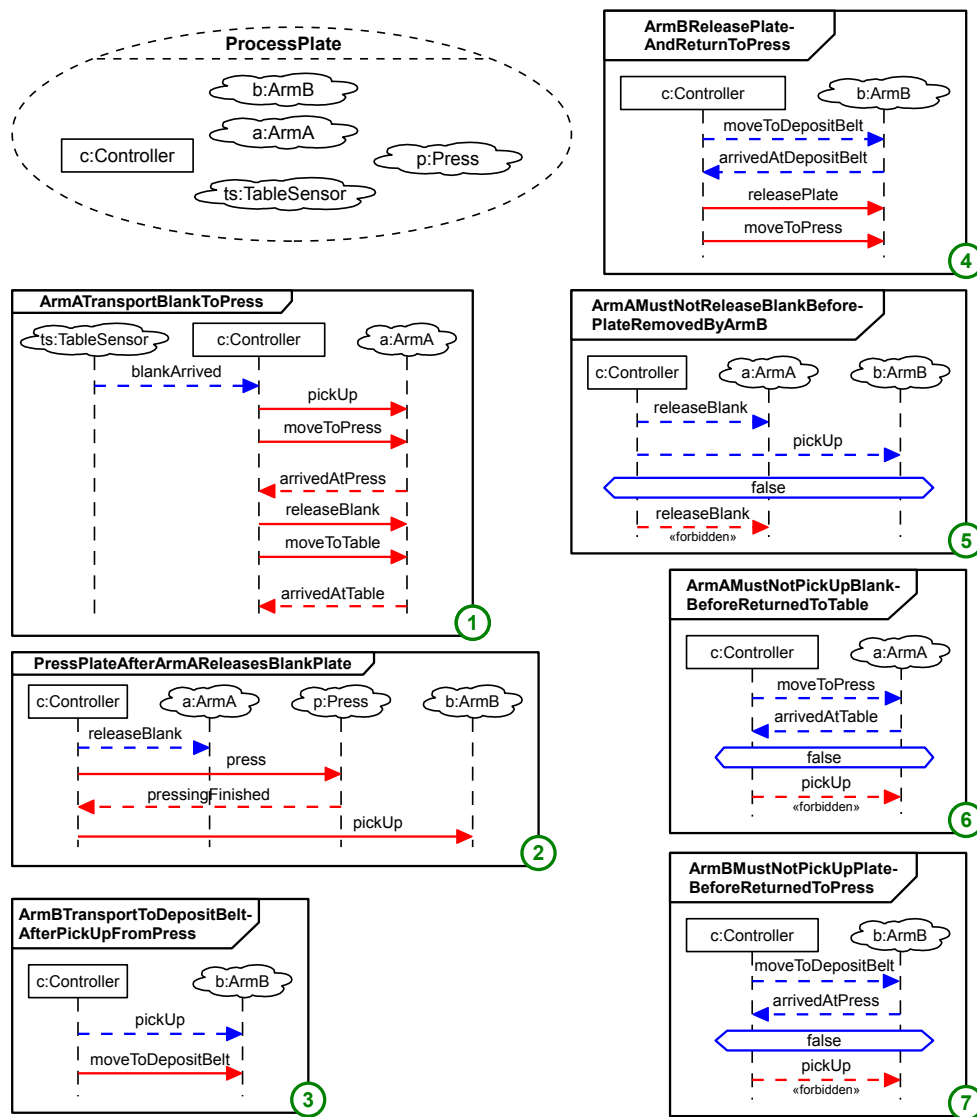


Figure C.26: The MSDs that specify the requirements of the production cell.

resembles a refined assumption of the environment behavior that an engineer might have added to capture the behavior of the robot arms more precisely.

The assumptions for **b:ArmB** works accordingly (modeled in the MSDs **ArmB-MoveFromPressToDepositBeltTimeAssumption** (11) and **ArmBMoveFromDepositBeltToPressTimeAssumption** (11)). The assumed behavior of the press is modeled in the MSD **PressPlateAssumption** (12), which says that the pressing process will be finished within a certain time interval. The length of the time intervals is represented by the parameters **FMIN**, **AMIN/AMAX**, **BMIN/BMAX**, and **PMIN/PMIN**. See the table in Fig. C.32 for concrete values that are given to the parameters.

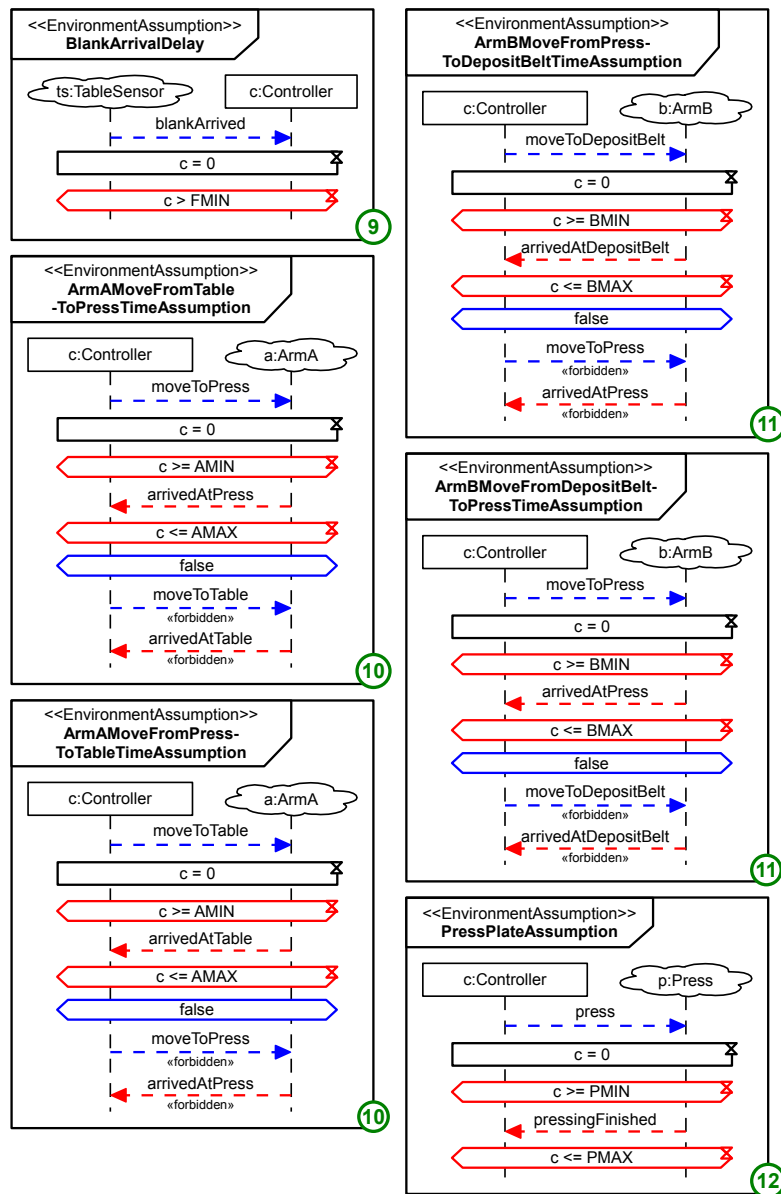


Figure C.27: The MSDs that specify the assumptions for the environment of the production cell.

Synthesis results

The tables in Fig. C.29 and C.28 show how long transformation and synthesis take for the production cell specification. Fig. C.28 shows the times of the synthesis in a setting where the system is allowed to delay active system event. The table in Fig. C.29 shows the synthesis times of the synthesis in a setting where the system is not allowed to do so (see Sect. 4.7.1).

The rows show the synthesis times and whether the synthesis reported the specification to be consistently executable or not for different concrete values for the delay parameters. Remember that checking consistent executability means that we consider a setting where the system can only execute steps that are currently active in an MSD.

The bottom of the table in Fig. C.28 shows the number of MSDs, lifelines, messages, clock variables, and time conditions in the specification. Furthermore, table shows the time for transforming the MSD specification and the number of TGG rules applied for the transformation.

The results are discussed in more detail at the end of Sect. C.3.3; Sect. C.4.1 explains in more detail how the measurements were taken.

	FMIN	AMIN	AMAX	BMIN	BMAX	PMIN	PMAX	consistently executable?	Time for synth. AG cond. (min)*	Time for synth. AGAF cond. (min)**
1	8	2	3	2	3	1	2	yes	34,12 min	mem out (after 7 hours)
2	0	2	3	2	3	1	2	no	0,07 min	mem out (after 7 hours)
3	1	2	3	2	3	1	2	no	0,04 min	aborted (after 12 hours)
4	5	2	3	2	3	1	2	no	11,80 min	11 hours
5	8	2	4	2	3	1	2	no	76,22 min	aborted (after 12 hours)
6	8	2	3	2	4	1	2	no	64,56 min	aborted (after 12 hours)
7	8	2	3	2	3	1	6	no	40,91 min	mem out (after 6 hours)
8	8	2	3	2	3	1	8	no	34,87 min	aborted (after 12 hours)

Number of MSDs	13
Number of lifelines	30
Number of messages	45
Number of clock variables	6
Number of time conditions	11
Number of TGG rules applied	114
Time for TGG trans (sec.)**	34

* Times measured on a machine with 5 Intel Xenon E5520 CPUs at 2,27Ghz and 72 GB RAM, Suse Linux (only one core and 4GB RAM used by Uppaal Tiga)
** Time measured on a laptop with one Intel Core2 T5500 CPU at 1,66Ghz and 2 GB RAM, running Windows XP

Figure C.28: Times of transformation and synthesis for the production cell specification with different concrete values for delays in the specification, in a setting where the system *is* allowed to delay active events.

C.3.3 Compositional synthesis of the production cell specification

This section shows how the production cell specification can be decomposed into two part specifications according to the compositional synthesis technique presented in Sect. 4.6. See also Sect. 4.6.3 for an explanation of how the two

	FMIN	AMIN	AMAX	BMIN	BMAX	PMIN	PMAX	consistently executable?	Time for synth. AG cond. (min)*	Time for synth. AGAF cond. (min)**
1	8	2	3	2	3	1	2	yes	0,04 min	0,09 min
2	0	2	3	2	3	1	2	no	0,01 min	1,19 min
3	1	2	3	2	3	1	2	no	0,02 min	1,37 min
4	5	2	3	2	3	1	2	no	0,07 min	1,41 min
5	8	2	4	2	3	1	2	no	0,05 min	1,16 min
6	8	2	3	2	4	1	2	no	0,05 min	1,29 min
7	8	2	3	2	3	1	6	no	0,08 min	1,76 min
8	8	2	3	2	3	1	8	no	0,10 min	2,06 min

* Times measured on a machine with 5 Intel Xenon E5520 CPUs at 2.27Ghz and 72 GB RAM, Suse Linux (only one core and 4GB RAM used by Uppaal Tiga)

** Time measured on a laptop with one Intel Core2 T5500 CPU at 1,66Ghz and 2 GB RAM, running Windows XP

Figure C.29: Times of transformation and synthesis for the production cell specification with different concrete values for delays in the specification, in a setting where the system is *not* allowed to delay active events.

part specifications are created. This section just briefly describes the part specifications with the additional assume/guarantee property that is added.

As described in Sect. 4.6.3, “step 0” of the decomposition of the specification is the decomposition of the system object `c:Controller` into the objects `c1:Controller1` and `c2:Controller2`. The idea is that `c1:Controller1` is responsible for controlling `a:ArmA` and `c2:Controller2` is responsible for controlling `b:ArmB` and `p:Press` (see also Fig. 4.14). The following two part specifications, called Transport Blank To Press and Press Plate And Transport To Deposit Belt, are created from the global production cell specification.

Part specification one Transport Blank To Press

The first part of the specification is shown in Fig. C.30. The collaboration diagram shows that this part only describes the behavior of the system instance `c1:Controller1` and the environment instances `ts:TableSensor` and `a:ArmA`. The MSDs `ArmATransportBlankToPress` (1) and `ArmAMustNotPickUpBlankBeforeReturnedToTable` (6) describe the requirements of `c1` and the assumption MSDs `BlankArrivalDelay` (9) as well as `ArmAMoveFromPressToTableTimeAssumption` (10) and `ArmAMoveFromTableToPressTimeAssumption` (10) describe the assumed behavior of the arriving blanks as well as the robot arm A. Additionally, the requirement MSD `BlankArrivalAtPressDelay` is introduced here. It states that blank plates must not be released into the press in intervals shorter than `RMIN`. (This MSD appears as an assumption in the second part specification Press Plate And Transport To Deposit Belt.)

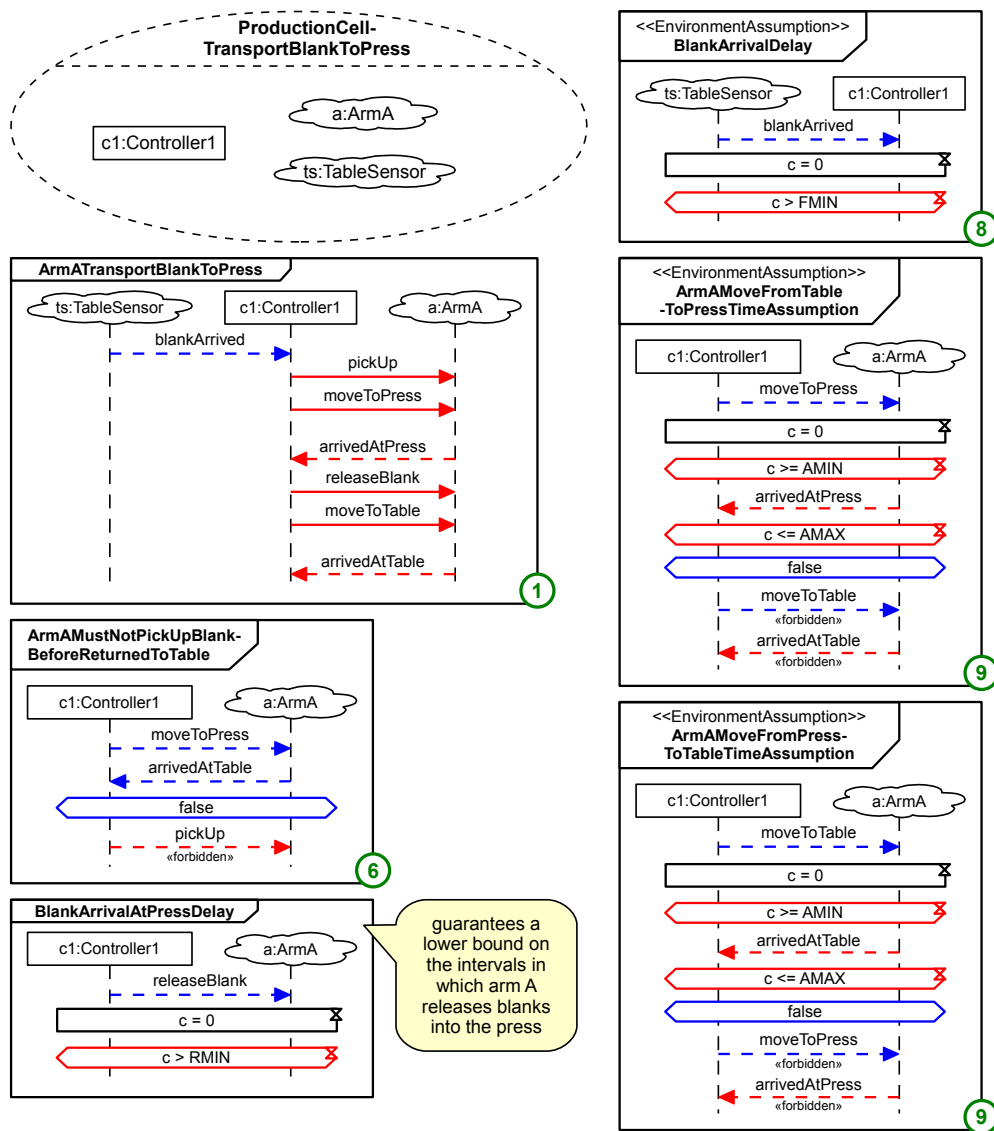


Figure C.30: Part one of the decomposed specification – controller `c1` controls arm A.

Part specification two Press Plate And Transport To Deposit Belt

The second part specification describes the requirements for the controller c_2 . The only instance not relevant in this part specification is the table sensor. The controller c_1 appears in this specification as an environment instance.

The requirements of this part specification are described by the MSDs (2)-(5) as already shown above. The only difference is that the MSDs `PressPlate-AfterArmAReleasesBlankPlate` (2) and `ArmAMustNotReleaseBlankBeforePlate-RemovedByArmB` (5) now have two lifelines representing the controllers c_1 and c_2 (see also Fig. 4.15 and Sect. 4.6.2, “Step 0”).

The assumption MSDs are the MSDs (10)-(11) as already introduced above. In addition this specification contains the assumption MSD `BlankArrivalAtPress-Delay`. This assumption MSD is the same MSD as the one introduced as an additional requirement (guarantee) in the part specification `Transport Blank To Press`.

Synthesis results

The tables in Fig. C.32 and C.33 show the synthesis results and times for the two part specification with different concrete values for the parameters `FMIN`, `RMIN` `AMIN/AMAX`, `BMIN/BMAX`, and `PMIN/PMIN`. (Section C.4.1 explains how the measurements were taken.) The table in Fig. C.32 shows the time of the synthesis in a setting where the system is allowed to delay active system event. The table in Fig. C.33 shows the synthesis times of the synthesis in a setting where the system is not allowed to do so (see Sect. 4.7.1).

The synthesis was tested with the following concrete values for the parameters:

- **Row 1:** the specification is consistently executable for these values
- **Row 2-4:** the time interval in which blanks may arrive at the table is too short,
- **Row 5:** the time that arm `a:ArmA` may take to move from the table to the press and from the press to the table, respectively, is too long.
- **Row 6:** the time that arm `b:ArmB` may take to move from the press to the deposit belt and from the deposit belt to the press, respectively, is too long.
- **Row 7-8:** the time that the press may take to process metal blanks to plates is too long.
- **Row 9:** shows that the first part specification will not be consistently executable if the *assume/guarantee property is too strong*.
- **Row 10:** shows that the second part specification will not be consistently executable if the *assume/guarantee property is too weak*.

The bottom of the table in Fig. C.32 shows the number of MSDs, lifelines, messages, clock variables, and time conditions per part specification. Furthermore, the time for transforming the MSD specification and the number of TGG rules applied for the transformation are displayed.

The *result* of this evaluation is, first, that *the synthesis produces correct results*, i.e., it reports a (part) specification to be consistently executable if it is expected to be consistently executable, and it reports that a (part) specification

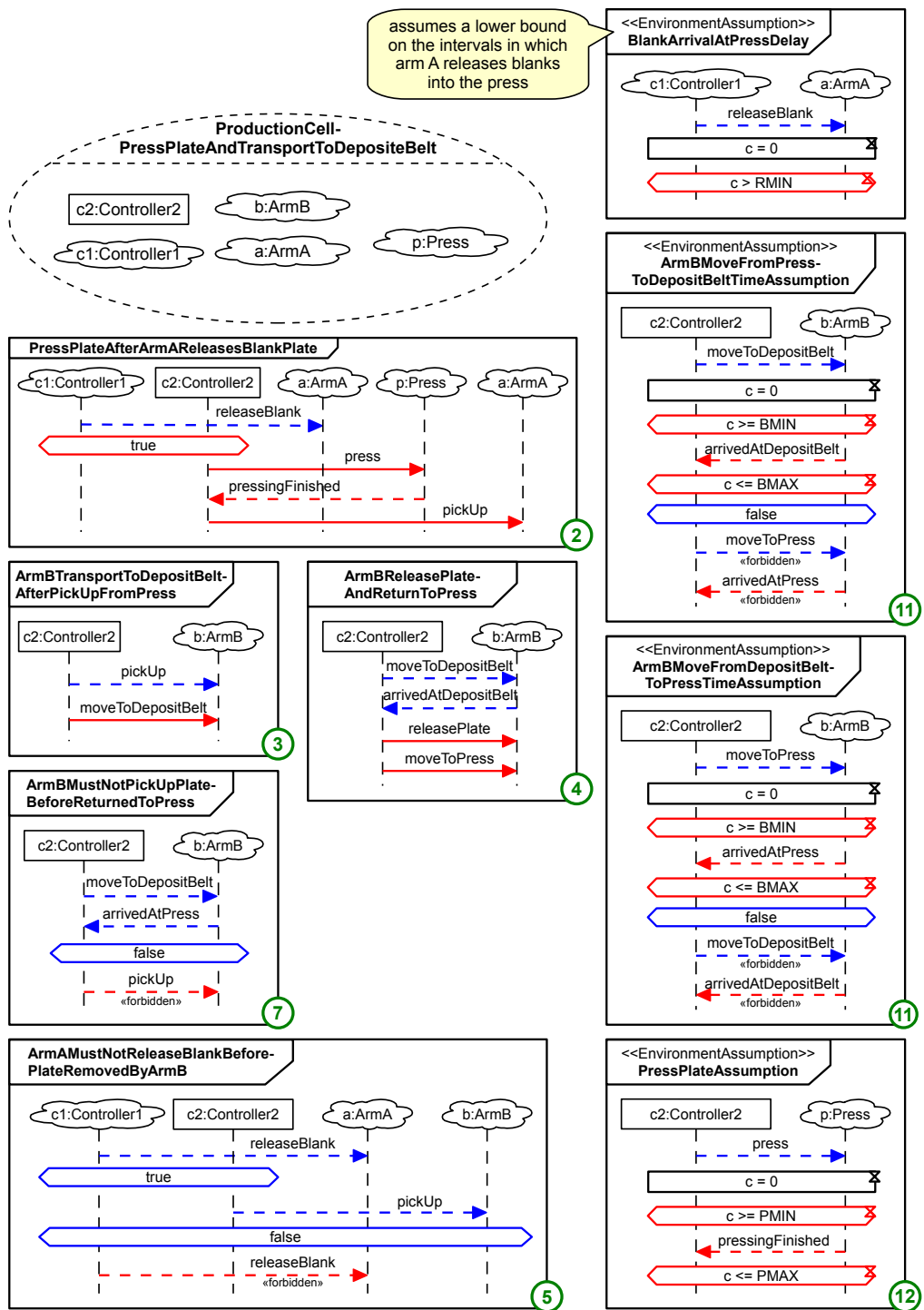


Figure C.31: Part two of the decomposed specification – controller *c2* controls the press and arm B.

	FMIN	RMIN	AMIN	AMAX	BMIN	BMAX	PMIN	PMAX	Part spec. I consistently executable?	Time for synth. AG cond. (min)*	Part spec. II consistently executable?	Time for synth. AG cond. (min)*	Time for synth. AGAF cond. (min)**	
1	8	7	2	3	2	3	1	2	yes	0.01	0.05	yes	0.09	2.03
2	0	7	2	3	2	3	1	2	no	0	0.02	yes	(as in 1)	(as in 1)
3	1	7	2	3	2	3	1	2	no	0	0.02	yes	(as in 1)	(as in 1)
4	5	7	2	3	2	3	1	2	no	0	0.03	yes	(as in 1)	(as in 1)
5	8	7	2	4	2	3	1	2	no	0	0.05	yes	(as in 1)	(as in 1)
6	8	7	2	3	2	4	1	2	yes	(as in 1)	(as in 1)	no	0.1	2.06
7	8	7	2	3	2	4	1	6	yes	(as in 1)	(as in 1)	no	0.08	1.71
8	8	7	2	3	2	4	1	8	yes	(as in 1)	(as in 1)	no	0.11	2.04
9	8	3	2	3	2	3	1	2	yes	0.01	0.04	no	0.1	1.18
10	8	9	2	3	2	3	1	2	no	0.01	0.05	yes	0.11	2.12

Number of MSDs	6	9	Sum:	15
Number of lifelines	13	22		36
Number of messages	20	27		47
Number of clock variables	4	4		8
Number of time conditions	6	7		13
Number of TGG rules applied				131
Time for TGG trans (ms)				31.5

* Times measured on a machine with 5 Intel Xeon E5520 CPUs at 2.27Ghz and 72 GB RAM. Suse Linux (only one core and 4GB RAM used by Uppaal TIGA)

** Time measured on a laptop with one Intel Core2 T5500 CPU at 1.66Ghz and 2 GB RAM, running Windows XP

Figure C.32: Times of transformation and synthesis for the part specifications of the production cell with different concrete values for delays in the specification, in a setting where the system is allowed to delay active events.

	FMN	RMN	AMN	AMAX	BMIN	BMAX	PMN	PMAX	Part spec. I consistently executable?	Time for synth. AG cond. (mn)*	Part spec. II consistently executable?	Time for synth. AG cond. (mn)*	Time for synth. AGAF cond. (mn)*	Time for synth. AGAF cond. (mn)*
1	8	7	2	5	2	3	1	2	yes	0	0	0.01	0.01	0.01
2	0	7	2	3	2	3	1	2	no	0	0.02	yes	(as in 1)	(as in 1)
3	1	7	2	3	2	3	1	2	no	0	0.01	yes	(as in 1)	(as in 1)
4	5	7	2	3	2	3	1	2	no	0	0.02	yes	(as in 1)	(as in 1)
5	8	7	2	4	2	3	1	2	no	0	0.02	yes	(as in 1)	(as in 1)
6	8	7	2	3	2	4	1	2	yes	(as in 1)	(as in 1)	no	0.01	0.05
7	8	7	2	3	2	4	1	6	yes	(as in 1)	(as in 1)	no	0.01	0.05
8	8	7	2	3	2	4	1	8	yes	(as in 1)	(as in 1)	no	0.01	0.04
9	8	3	2	3	2	3	1	2	yes	(as in 1)	(as in 1)	no	0.01	0.04
10	8	9	2	3	2	3	1	2	no	0	0.02	yes	0.01	0.01

* Times measured on a machine with 5 Intel Xenon E5520 CPUs at 2.27Ghz and 72 GB RAM, Suse Linux (only one core and 4GB RAM used by Uppaal Tiga)
 ** Time measured on a laptop with one Intel Core2 T5500 CPU at 1.66Ghz and 2 GB RAM, running Windows XP

Figure C.33: Times of transformation and synthesis for the part specifications of the production cell with different concrete values for delays in the specification, in a setting where the system is *not* allowed to delay active events.

is not consistently executable if it can be expected to be not consistently executable. The results are correct using both the AG or AGAF winning condition.

Second, it turns out that *disallowing the system to delay the execution of active events can drastically reduce the synthesis time.*

Third, the *compositional synthesis can greatly reduce the complexity of the synthesis problem.* For example, synthesizing a controller for the global specification was not possible when using the AGAF winning condition and allowing the system to delay the execution of active events. The synthesis aborted because the available memory was used up after three to seven hours. Synthesizing controllers for the part specifications in sum takes about 2 minutes (the memory used was not measured).

C.4 Synthesis performance measurement

This section presents results from measuring the performance of the synthesis technique that developed in the scope of this thesis. The synthesis time was measured by the help of technical example MSD specifications, which are of such a kind that with a growing number of MSDs in the specification, the induced state space grows exponentially. The synthesis times are furthermore compared to the times for the TGG transformation, which maps the UML-based MSD specifications to the corresponding TGA system.

The main results of these measurements are

- The synthesis of controllers for untimed MSD specifications can be carried out within an acceptable time, i.e., in less than one minute, for specifications that describe more than 3^{10} many reachable super-cuts.
- For untimed MSD specifications, the complexity of the synthesis grows linearly with the state space induced by the MSD specification.
- For timed MSD specifications, the synthesis greatly suffers from the number of interrelated time intervals.
- The speed of the TGG transformation grows linearly in the size of the MSD specifications.

C.4.1 How the measurements were taken

The the MSD-to-TGA transformation and the synthesis by UPPAAL TIGA were executed and measured on two different machines with the following configurations:

- **MSD-to-TGA transformation:** Intel Core2 processor (T5500) at 1,66GHz and 2GB RAM, Windows XP, version 0.3 of the TGG INTERPRETER. The TGG used for the transformation is the one documented in Appendix B.
- **Uppaal TIGA synthesis:** Intel Xenon E5520 at 2,27Ghz and 72 GB RAM³, SUSE Linux (2.6.27.45-0.1), using UPPAAL TIGA is 0.14 (rev. 4530), May 2010, as included in the ECDAR tool⁴, version 0.8.

C.4.2 Untimed specifications with exponentially growing state space

The first measurement relies on a number of example MSD specifications that have the following form as shown in Fig. C.34. Each MSD specification consists of n MSDs where the first MSD is triggered by an environment event `do` and then triggers two activations of the second MSD. Subsequently, this MSD triggers two activations of the third MSD, etc. Fig. C.34 shows the MSD specification for $n = 3$. This leads to a cascade of subsequent activations of MSDs and the state space described by the MSD specification grows exponential with respect to the number of MSDs in it. Because each MSD can be in three different cuts, we approximate the state space with 3^n .

³of which UPPAAL TIGA is only able to address 4 GB

⁴<http://www.cs.aau.dk/~adavid/ecdar/>

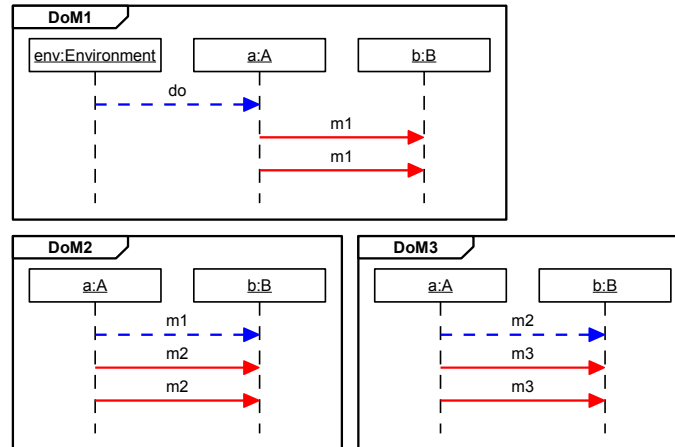


Figure C.34: Example of an untimed MSD specification with exponentially many activations of MSDs (here $n = 3$)

The table in Fig. C.35 shows the results from transforming and synthesizing controllers from the MSD specifications with one to thirteen MSDs. The first three columns show the number of MSDs per specification as well as the number of lifelines and the number of messages. The last two columns show the number of TGG rules that were applied for transforming the specification into a TGA model as well as the time the transformation took. The fourth and fifth column show the time it took UPPAAL TIGA for synthesizing a strategy satisfying the *AGAF* and *AG* properties, see Cond. 4.2 and 4.1 in Sect. 4.4.

	No. of MSDs	No. of lifelines	No. of messages	Time for synth. AG cond. (min)*	Time for synth. AGAF cond. (min)*	No. of TGG rules applied	Time for TGG transformation (sec)**
1	3	3	0	0	0	9	1,2
2	5	6	0	0	0	15	2,5
3	7	9	0	0	0	21	3,3
4	9	12	0	0	0	27	4,4
5	11	15	0	0	0	33	5,7
6	13	18	0	0	0	39	7
7	15	21	0,01	0,01	0,01	45	8,1
8	17	24	0,02	0,05	0,02	51	9,1
9	19	27	0,08	0,2	0,02	57	10,5
10	21	30	0,28	0,74	0,03	63	11,8
11	23	33	1,03	2,8	0,03	69	13
12	25	36	3,68	10,21	0,03	75	14,6
13	27	39	mem out	mem out	0,03	81	15,9

* Times measured on a machine with 5 Intel Xenon E5520 CPUs at 2,27Ghz and 72 GB RAM, Suse Linux (only one core and 4GB RAM used by Uppaal TIGA)

** Time measured on a laptop with one Intel Core2 T5500 CPU at 1,66Ghz and 2 GB RAM, running Windows XP

Figure C.35: A table listing the time for transforming and synthesizing strategies for an untimed MSD specification with exponentially many MSD activations.

Strategies can be successfully synthesized from specifications with up to twelve MSDs. For bigger specifications, the synthesis algorithm runs out of memory (4 GB) after a few minutes. The TGG transformation times grow linearly with the size of the MSD specification. An additional MSD takes the transformation one additional second to translate. Up to the size of 8 MSDs, the synthesis is faster than the transformation, but then quickly suffers from the state space explosion. The chart in Fig. C.36 illustrates this graphically.

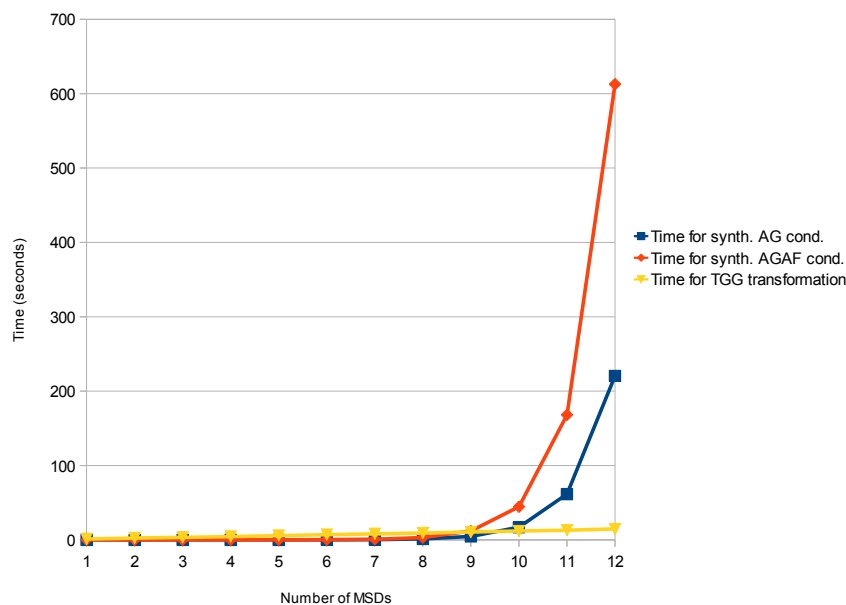


Figure C.36: A chart visualizing the time for transforming and synthesizing strategies for an untimed MSD specification with exponentially many MSD activations.

The synthesis times grow linearly with the state space described by the specification. The chart in Fig. C.37 illustrates this by plotting the size of the MSD specification against the logarithm of the synthesis times.

Cassez et al. prove that the synthesis algorithm for synthesizing strategies for untimed reachability/safety games in UPPAAL TIGA is efficient, i.e., at most by a constant factor slower than the fastest possible synthesis algorithm. The synthesis time is linear to the state space of the given synthesis problem [CDF⁺05]. The measurements suggest that also the synthesis of controllers from MSD specifications by the MSD-to-TGA mapping presented in Chap. 4 is efficient in the sense that it is not more than a constant factor slower than the fastest possible synthesis algorithm.

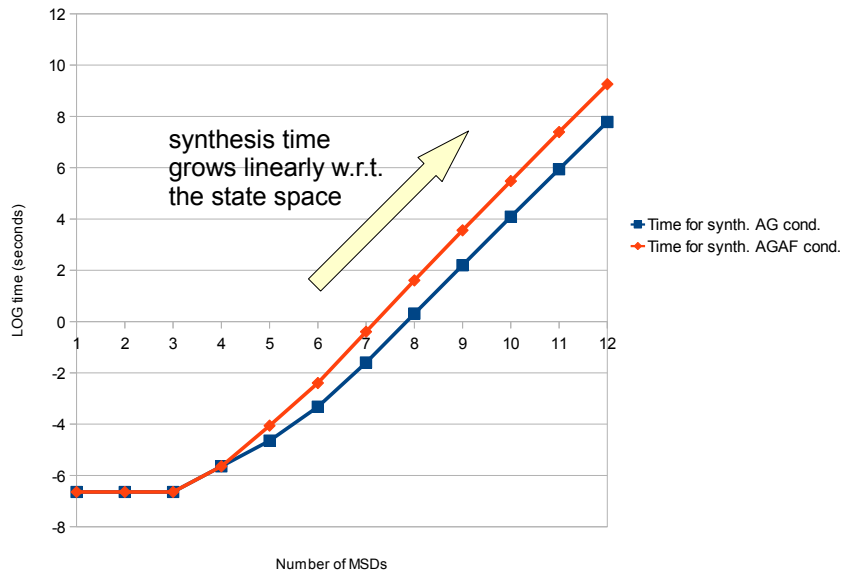


Figure C.37: Plotting the logarithm (\log_2) of the synthesis time against the size of the MSD specification

C.4.3 Timed specification with exponentially growing state space

To measure the performance of the synthesis for timed specifications, MSD specifications were considered that also follow a similar schema as the specifications presented in Sect. C.4.2. In the kinds of specifications considered here, an additional clock reset and a minimal and maximal delay is added to each MSD. With each additional MSD, the time for the system to complete all the MSDs grows exponentially. Figure C.38 shows the specification with 5 MSDs. Each MSD specifies a minimal delay of 1 and a maximal delay that depends on the number of diagrams that are going to be subsequently triggered by the MSD. One of the MSDs is an assumption MSD that prohibits the environment sending another `do` event too early. If this assumption MSD was not included in the specification, the specification could be easily violated.

The measurements show that synthesizing strategies for this kind of timed specifications takes significantly longer. Synthesizing a strategy satisfying the *AGAF* condition from a specification with 6 MSDs (including the one assumption MSD) already takes over eight minutes. Synthesizing a strategy satisfying the *AG* condition from a specification with 7 MSDs takes over 26 minutes. Synthesizing a strategy satisfying the *AGAF* condition from a specification with 7 MSDs was successful only after over 10 hours. The chart shown in Fig. C.40 visualizes the synthesis and transformation times.

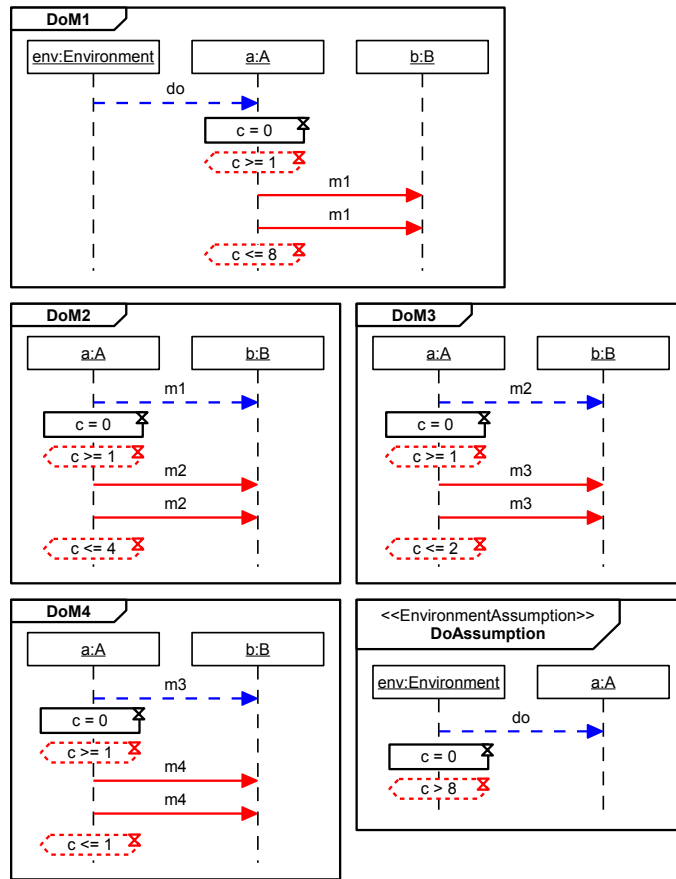


Figure C.38: Example of a timed MSD specification with exponentially many activations of MSDs (here $n = 4$)

	No. of MSDs	No. of lifelines	No. of messages	No. of clock variables	No. of time conditions	Time for synth. AG cond. (min)*	Time for synth. AGAF cond. (min)*	No. of TGG rules applied	Time for TGG trans (sec.)**
2	5	4	2	3	0	0	18	2,1	
3	7	7	3	5	0	0	27	3,7	
4	9	10	4	7	0	0	36	5,5	
5	11	13	5	9	0,04	0,42	45	7,1	
6	13	16	6	11	0,69	8,72	54	8,7	
7	15	19	7	13	26,58	10 hours	63	10,3	

* Times measured on a machine with 5 Intel Xenon E5520 CPUs at 2,27Ghz and 72 GB RAM, Suse Linux (only one core and 4GB RAM used by Uppaal Tiga)

** Time measured on a laptop with one Intel Core2 T5500 CPU at 1,66Ghz and 2 GB RAM, running Windows XP

Figure C.39: A table listing the time for transforming and synthesizing strategies for a timed MSD specification with exponentially many MSD activations.

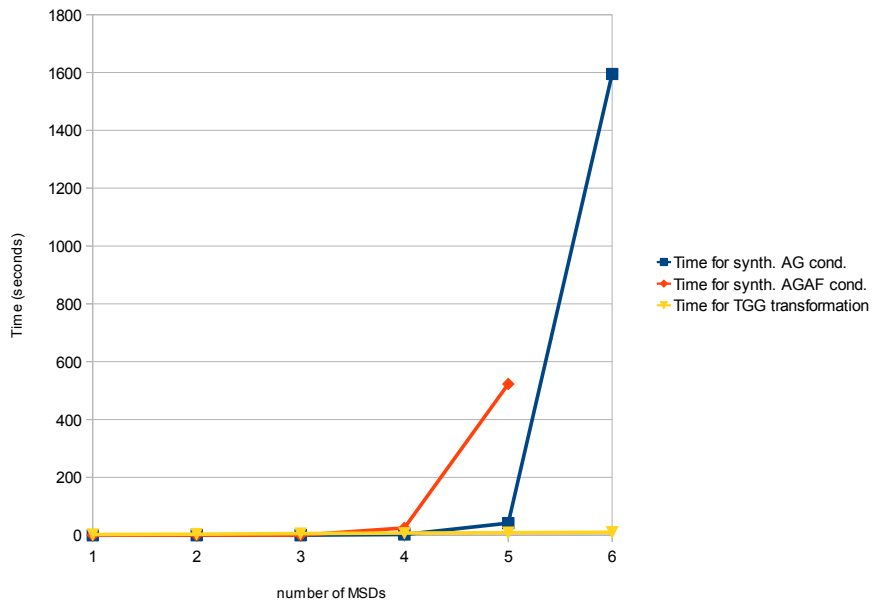


Figure C.40: A chart visualizing the time for transforming and synthesizing strategies for a timed MSD specification with exponentially many MSD activations.

The measurements show that the synthesis suffers greatly from the many interrelated time intervals that are described by the specification. The measurements suggest that the synthesis time grows more than linearly with the state space that is described by the MSD specification, but there are too few measurements to identify any correlations between the specification size and the synthesis times.

Bibliography

- [ACD93] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-Checking in Dense Real-time. *Information and Computation*, 104:2–34, 1993.
- [AD94] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
- [ADG⁺09] Philipp Adelt, Jörg Donoth, Jürgen Gausemeier, Jens Geisler, Stefan Henkler, Sascha Kahl, Benjamin Klöpper, Alexander Krupp, Eckehard Münch, Simon Oberthür, Carlos Paiz, Mario Porrmann, Rafael Radkowski, Christoph Romaus, Alexander Schmidt, Bernd Schulz, Henner Vöcking, Ulf Witkowski, Katrin Witting, and Oleksiy Znamenshchykov. *Selbstoptimierende Systeme des Maschinenaubaus – Definitionen, Anwendungen, Konzepte*, volume 234. HNI-Verlagsschriftenreihe, Heinz Nixdorf Institut Paderborn, August 2009. (in German).
- [AKRS06] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066 of *Lecture Notes in Computer Science (LNCS)*, pages 361–375, Heidelberg, 2006. Springer Verlag.
- [BB91] Albert Benveniste and Gérard Berry. The Synchronous Approach to Reactive and Real-Time Systems. *Proceedings of the IEEE, special section "Another look at Real-time programming"*, 79(9):1270–1282, September 1991.
- [Büc66] J. Richard Büchi. Symposium on decision problems: On a decision method in restricted second order arithmetic. In Patrick Suppes Ernest Nagel and Alfred Tarski, editors, *Logic, Methodology and Philosophy of Science, Proceeding of the 1960 International Congress*, volume 44 of *Studies in Logic and the Foundations of Mathematics*, pages 1–11. Elsevier, 1966.
- [BCD⁺06] Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. UPPAAL-Tiga: Timed Games for Everyone. In Luca Aceto and Anna Ingólfóttir, editors,

- Proceedings of the 18th Nordic Workshop on Programming Theory (NWPT'06)*, Reykjavik, Iceland. Reykjavik University, 2006.
- [BCD⁺07a] Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. UPPAAL-Tiga: Time for Playing Games! In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 121–125, Berlin, Germany, July 2007. Springer.
- [BCD⁺07b] Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Uppaal Tiga User-manual. online resource, 2007. <http://www.cs.aau.dk/~adavid/tiga/manual.pdf> – last accessed: April 2011.
- [BCS00] Francis Bordeleau, Jean-Pierre Corriveau, and Bran Selic. A scenario-based approach to hierarchical state machine design. In *Proceedings of the third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, pages 78–85, 2000.
- [BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *LNCS, Formal Methods for the Design of Real-Time Systems (revised lectures)*, volume 3185, pages 200–236. Springer Verlag, 2004.
- [BG03] Sven Burmester and Holger Giese. The Fujaba Real-Time State-chart Plugin. In *Proceedings of the first International Fujaba Days 2003, Kassel, Germany, 2003*.
- [BH02] Yves Bontemps and Patrick Heymans. Turning High-Level Live Sequence Charts into Automata. In *Proceedings of the 1st Workshop on Scenarios and State Machines: Models Algorithms and Tools (SCESM'02), 24th International Conference on Software Engineering (ICSE'02), May 2002*. ACM, 2002.
- [BH05] Yves Bontemps and Patrick Heymans. From Live Sequence Charts to State Machines and Back: A Guided Tour. *IEEE Transactions on Software Engineering*, 31(12):999–1014, 2005.
- [BJ95] Manfred Broy and Stefan Jähnichen, editors. *KORSO: Methods, Languages, and Tools for the Construction of Correct Software: Final Report*. Springer, 1995.
- [BLL⁺96] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL – a Tool Suite for Automatic Verification of Real-Time Systems. In *Proceedings of the DIMACS/SYCON workshop on Hybrid Systems III: Verification and Control*, pages 232–243, 1996.
- [BS03] Yves Bontemps and Pierre-Yves Schobbens. Synthesis of Open Reactive Systems from Scenario-Based Specifications. In *Proceedings of the 3rd International Conference on Application of Concurrency*

- to *System Design (ACSD 2003)*, 18-20 June 2003, Guimaraes, Portugal, pages 41–50, 2003.
- [BS05] Yves Bontemps and Pierre-Yves Schobbens. The Complexity of Live Sequence Charts. In Sassone, editor, *Proceedings of the International Conference on Foundations of Software Science and Computation Structure (FoSSACS'05)*, pages 364–378, April 2005.
- [BSL04] Yves Bontemps, Pierre Yves Schobbens, and Christof Löding. Synthesis of Open Reactive Systems from Scenario-Based Specifications. *Fundamenta Informaticae*, 62(2):139–169, 2004.
- [Cas07] Franck Cassez. Efficient on-the-fly algorithms for partially observable timed games. In *Proceedings of the 5th International Conference on Formal Modeling and Analysis of Timed Systems, FORMATS'07*, pages 5–24, Berlin, Heidelberg, 2007. Springer-Verlag.
- [CDF⁺05] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient On-the-fly Algorithms for the Analysis of Timed Games. *Proceedings of the 16th International Conference on Concurrency Theory (CONCUR'05)*, 3653:66–80, August 2005.
- [CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.
- [Dan09] Duc-Hanh Dang. *On Integrating Triple Graph Grammars and OCL for Model-Driven Development*. PhD thesis, University of Bremen, September 25th 2009.
- [DG08] Duc-Hanh Dang and Martin Gogolla. On Integrating OCL and Triple Graph Grammars. In *Models in Software Engineering: Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers*, 2008.
- [DH98] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. Technical Report CS98-09, Weizmann Institute of Science, Faculty of Mathematics and Computer Science, 1998.
- [DH01] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. In *Formal Methods in System Design*, volume 19, pages 45–80. Kluwer Academic Publishers, 2001. (Preliminary version in Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99), P. Ciancarini, A. Fantechi and R. Gorrieri, eds., Kluwer Academic Publishers, 1999, pp. 293-312.
- [DLDvL05] Christophe Damas, Bernard Lambeau, Pierre Dupont, and Axel van Lamsweerde. Generating Annotated Behavior Models From End-User Scenarios. *IEEE Transactions on Software Engineering*, 31(12):1056–1073, 2005.

- [DLvL06] Christophe Damas, Bernard Lambeau, and Axel van Lamsweerde. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*, pages 197–207. ACM, 2006.
- [FMS09] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann, 2009.
- [Fra06] Ursula Frank. *Spezifikationstechnik zur Beschreibung der Prinziplösung selbstoptimierender Systeme*. PhD thesis, Fakultät für Maschinenbau, Universität Paderborn, 2006. (in German).
- [GB03] Holger Giese and Sven Burmester. Real-Time Statechart Semantics. Technical Report tr-ri-03-239, Software Engineering Group, University of Paderborn, Paderborn, Germany, June 2003.
- [GB04] Holger Giese and Sven Burmester. Analysis and Synthesis for Parameterized Timed Sequence Diagrams. In *Proceedings of the 3rd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (ICSE 2003 Workshop W5S)*, Edinburgh, Scotland, 2004.
- [GBRP10] J. Gausemeier, R. Brandis, and M. Reyes-Perez. A Specification Technique for the Integrative Conceptual Design of Mechatronic Productions and Production Systems. In *Proceedings of the 11th International Design Conference DESIGN 2010*, 2010.
- [GDN10] Jürgen Gausemeier, Rafal Dorociak, and Alexander Nyßen. The mechatronic modeller: A software tool for computer-aided modeling of the principle solution of an advanced mechatronic system. In *11th International Workshop on Research and Education in Mechatronics*, September 2010.
- [GEK01] Jürgen Gausemeier, Peter Ebbesmeyer, and Ferdinand Kallmeyer. *Produktinnovation. Strategische Planung und Entwicklung der Produkte von morgen*. Carl Hanser Verlag, 2001. (in German).
- [GFDK08a] Jürgen Gausemeier, Ursula Frank, Jörg Donoth, and Sascha Kahl. Spezifikationstechnik zur Beschreibung der Prinziplösung selbstoptimierender Systeme des Maschinenbaus (Teil 1). *Konstruktion*, 7/8:59–66, July/August 2008. (in German).
- [GFDK08b] Jürgen Gausemeier, Ursula Frank, Jörg Donoth, and Sascha Kahl. Spezifikationstechnik zur Beschreibung der Prinziplösung selbstoptimierender Systeme des Maschinenbaus (Teil 2). *Konstruktion*, 9:91–99, September 2008. (in German).
- [GFDK09] Jürgen Gausemeier, Ursula Frank, Jörg Donoth, and Sascha Kahl. Specification Technique for the Description of Self-Optimizing Mechatronic Systems. *Research in Engineering Design*, 20(4):201–223, 2009.

- [GHHK06] Holger Giese, Stefan Henkler, Martin Hirsch, and Florian Klein. Nobody's perfect: Interactive Synthesis from Parametrized Real-Time Scenarios. In *Proceedings of the 5th ICSE 2006 Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06), Shanghai, China, 2006*.
- [Gie03] Holger Giese. Towards Scenario-Based Synthesis for Parametric Timed Automata. In *Proceedings of the 2nd International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM, ICSE 2003 Workshop 8), Portland, USA, 2003*.
- [GK07] Joel Greenyer and Ekkart Kindler. Reconciling TGGs with QVT. In G. Engels, B. Opdyke, D.C. Schmidt, and Weil, F. (Eds.), editors, *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2007, September 30 - October 5, 2007, Nashville, USA, LNCS, volume Volume 4735, pages pp. 16–30, Berlin, 2007*. Springer Verlag.
- [GK10] Joel Greenyer and Ekkart Kindler. Comparing relational model transformation technologies: implementing Query/View/Transformation with Triple Graph Grammars. *Software and Systems Modeling (SoSyM)*, 9(1):21–46, January 2010. Published online July 15, 2009.
- [GKB05] Holger Giese, Florian Klein, and Sven Burmester. Pattern Synthesis from Multiple Scenarios for Parameterized Real-Timed UML Models. In *Scenarios: Models, Algorithms and Tools*. Springer Verlag, 2005.
- [GKS00] Radu Grosu, Ingolf Krüger, and Thomas Stauner. Hybrid sequence charts. In *ISORC '00: Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 104, Washington, DC, USA, 2000. IEEE Computer Society.
- [GL94] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.
- [Gre06] Joel Greenyer. A study of technologies for model transformation: Reconciling TGGs with QVT. Diplomarbeit, Fakultät für Elektrotechnik, Informatik und Mathematik, Universität Paderborn, 2006.
- [Gre09] Joel Greenyer. Integrating Models for the Design of Mechatronic Systems. In *Proceedings des gemeinsamen Workshops der Informatik-Graduiertenkollegs und Forschungskollegs*, pages 173–174, Dagstuhl, June 2009.
- [Gre10] Joel Greenyer. Synthesizing Modal Sequence Diagram Specifications with Uppaal-Tiga. Technical Report tr-ri-10-310, University of Paderborn, February 2010.

- [GSG⁺09] Jürgen Gausemeier, Wilhelm Schäfer, Joel Greenyer, Sascha Kahl, Sebastian Pook, and Jan Rieke. Management of Cross-Domain Model Consistency During the Development of Advanced Mechatronic Systems. In Margareta Norell Bergendahl, Martin Grimheden, and Larry Leifer, editors, *Proceedings of the 17th International Conference on Engineering Design (ICED'09)*, volume 6, pages 1–12, University of Stanford, CA, USA, August 2009. Design Society.
- [GT05] Holger Giese and Sergej Tissen. The SceBaSy Plugin for the Scenario-Based Synthesis of Real-Time Coordination Patterns for Mechatronic UML. In *Proceedings of the 3rd International Fujaba Days 2005, Paderborn, Germany, 2005*.
- [GZD⁺08] Jürgen Gausemeier, Detmar Zimmer, Jörg Donoth, Sebastian Pook, and Alexander Schmidt. Proceeding for the Conceptual Design of Self-Optimizing Mechatronic Systems. In *Proc. 10th International Design Conference*, Dubrovnik, Croatia, May 19-22 2008.
- [HB10] Sylvain Halle and Tevfik Bultan. Realizability Analysis for Message-based Interactions Using Shared-State Projections. In *Proceedings of the 18th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, FSE 2010, Santa Fe, New Mexico, November 7-11, 2010*, 2010.
- [HGH⁺09] Stefan Henkler, Joel Greenyer, Martin Hirsch, Wilhelm Schafer, Kahtan Alhawash, Tobias Eckardt, Christian Heinzemann, Renate Löffler, Andreas Seibel, and Holger Giese. Synthesis of Timed Behavior From Scenarios in the Fujaba Real-Time Tool Suite. In *Proceedings of the 31th International Conference on Software Engineering (ICSE), Vancouver, Canada, 2009*.
- [HK99] David Harel and Hillel Kugler. Synthesizing State-Based Object Systems from LSC Specifications. Technical report, The Weizmann Institute of Science, Jerusalem, Israel, 1999.
- [HK02] David Harel and Hillel Kugler. Synthesizing State-Based Object Systems from LSC Specifications. In *International Journal of Foundations of Computer Science*, volume 13:1, pages 5–51, 2002. (Also, Proc. of the 5th International Conference on Implementation and Application of Automata (CIAA 2000; invited paper), Lecture Notes in Computer Science, Vol. 2088, Springer-Verlag, 2001, pp. 1-33.).
- [HKMP02a] David Harel, Hillel Kugler, Rami Marelly, and Amir Pnueli. Smart Play-Out of Behavioral Requirements. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2002, Portland, OR, USA, November 6-8, 2002*, pages 378–398, 2002.
- [HKMP02b] David Harel, Hillel Kugler, Rami Marelly, and Amir Pnueli. Smart Play-Out of Behavioral Requirements. Technical report, The Weizmann Institute of Science, 2002.

- [HKP04] David Harel, Hillel Kugler, and Amir Pnueli. Smart Play-Out Extended: Time and Forbidden Elements. In *Proceedings of the 4th International Conference on Quality Software (QSIC'04)*, pages 2–10. IEEE Computer Society, 2004.
- [HKP05] David Harel, Hillel Kugler, and Amir Pnueli. Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. In Hans-Jörg Kreowski, Ugo Montanari, Fernando Orejas, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Formal Methods in Software and Systems Modeling*, volume 3393/2005, pages 309–324. Springer-Verlag, 2005.
- [HM02a] David Harel and Rami Marelly. Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach. *Software and System Modeling (SoSyM)*, 2:2003, 2002.
- [HM02b] David Harel and Rami Marelly. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In *Proceedings of the 10th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'02)*, Fort Worth, Texas, pages 193–202, 2002.
- [HM03] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, August 2003.
- [HM08] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling (SoSyM)*, 7(2):237–252, May 2008.
- [HMS08] David Harel, Shahar Maoz, and Itai Segall. Some Results on the Expressive Power and Complexity of LSCs. In Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich, editors, *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, volume 4800 of *Lecture Notes in Computer Science*, pages 351–366. Springer, 2008.
- [HP85] David Harel and Amir Pnueli. On the Development of Reactive Systems. *Logics and Models of Concurrent Systems*, 13:477–498, 1985.
- [HS07] David Harel and Itai Segall. Planned and traversable play-out: a flexible method for executing scenario-based programs. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'07, pages 485–499, Berlin, Heidelberg, 2007. Springer-Verlag.
- [Ise05] Rolf Isermann. *Mechatronic Systems: Fundamentals*. Springer, 2005.
- [Ise07] Rolf Isermann. *Mechatronic Systems, Sensors, and Actuators: Fundamentals and Modeling*, chapter 2 Mechatronic Design Approach, pages 2–1 to 2–16. CRC Press, 2nd edition edition, November 2007. (1st edition 2002).

- [Ise09] Rolf Isermann. *Springer Handbook of Automation*, chapter 19 Mechatronic Systems—A Short Introduction, pages 317–331. Springer, July 2009.
- [ITU96] ITU-TS Recommendation Z.120: Message Sequence Charts (MSC), 1996.
- [Jac92] Ivar Jacobson. *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Professional, 1992.
- [Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.
- [Ka198] Ferdinand Kallmeyer. *Eine Methode zur Modellierung prinzipieller Lösungen mechatronischer Systeme*. PhD thesis, University of Paderborn, Heinz Nixdorf Institute, 1998. (in German).
- [KGSB99] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to Statecharts. In *DIPES '98: Proceedings of the IFIP WG10.3/WG10.5 International Workshop on Distributed and Parallel Embedded Systems*, pages 61–71, Norwell, MA, USA, 1999. Kluwer Academic Publishers.
- [KKS07] Felix Klar, Alexander Königs, and Andy Schürr. Model Transformation in the Large. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The foundations of Software Engineering, Dubrovnik, Croatia, Sept. 03 - 07, 2007*, pages 285 – 294, September 2007.
- [KPP09] Hillel Kugler, Cory Plock, and Amir Pnueli. Controller Synthesis from LSC Requirements. In Marsha Chechik and Martin Wirsing, editors, *Proceedings of the 12th International Conference of Fundamental Approaches to Software Engineering, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009.*, volume 5503 of *Lecture Notes in Computer Science*, pages 79–93. Springer, 2009.
- [Krü00] Ingolf Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technische Universität München, Institut für Informatik, 2000.
- [KS09] Hillel Kugler and Itai Segall. Compositional Synthesis of Reactive Systems from Live Sequence Chart Specifications. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009*, pages 77–91, 2009.
- [KSTM98] Kai Koskimies, Tarja Systa, Jyrki Tuomi, and Tatu Mannisto. Automated support for modeling oo software. *IEEE Software*, 15(1):87–94, 1998.

- [KTWW06] Jochen Klose, Tobe Toben, Bernd Westphal, and Hartmut Wittke. Check It Out: On the Efficient Formal Verification of Live Sequence Charts. In Thomas Ball and Robert B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification, CAV 2006, Seattle, WA, USA, August 17-20, 2006.*, volume 4144 of *Lecture Notes in Computer Science*, pages 219–233. Springer, 2006.
- [KW01] Jochen Klose and Hartmut Wittke. An Automata Based Interpretation of Live Sequence Charts. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy*, volume 2031 of *Lecture Notes In Computer Science*, pages 512–527. Springer, 2001.
- [KW07] Ekkart Kindler and Robert Wagner. Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical Report tr-ri-07-284, Department of Computer Science, University of Paderborn, June 2007.
- [LHLH01] Joachim Lückel, Thorsten Hestermeyer, and Xiaobo Liu-Henke. Generalization of the Cascade Principle in View of a Structured Form of Mechatronic Systems. In *Proceedings of the IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM 2001)*, Villa Olmo, Como, Italy, 2001.
- [LL94] Claus Lewerentz and Thomas Lindner, editors. *Formal Development of Reactive Systems*, volume 891 of *LNCS*. Springer, 1994.
- [LL06] Claus Lewerentz and Thomas Lindner. *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, chapter Case study “production cell”: A comparative study in formal specification and verification, pages 388–416. Springer, 2006.
- [LLNP09] Kim G. Larsen, Shuhao Li, Brian Nielsen, and Saulius Pusinskas. Scenario-based analysis and synthesis of real-time systems using Uppaal. In *Proceedings of the 13th Conference on Design, Automation, and Test in Europe (DATE'10)*. Aalborg University, 2009. Technical Report.
- [LLNP10] Kim G. Larsen, Shuhao Li, Brian Nielsen, and Saulius Pusinskas. Scenario-based analysis and synthesis of real-time systems using Uppaal. In *Proceedings of the 13th Conference on Design, Automation, and Test in Europe (DATE'10)*, March 2010.
- [LMR98] Stefan Leue, Lars Mehrmann, and Mohammad Rezai. Synthesizing ROOM Models from Message Sequence Chart Specifications. Technical report, University of Waterloo, Canada, 1998.
- [Loc07] Malte Lochau. On Synthesizing Statecharts from Live Sequence Chart Specifications. Master’s thesis, Technical University Carolo-Wilhelmina of Braunschweig, Institute for Programming and Reactive Systems, 2007.

- [LS98] X. Liu and S. Smolka. Simple Linear-Time Algorithm for Minimal Fixed Points. In *Proceedings of the 26th Conference on Automata, Languages and Programming (ICALP'98)*, volume 1443 of *LNCS*, pages 53–66, 1998.
- [Mao09] Shahar Maoz. Polymorphic Scenario-Based Specification Models: Semantics and Applications. In Andy Schürr and Bran Selic, editors, *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems, MODELS 2009, Denver, CO, USA, October 4-9, 2009.*, volume 5795 of *Lecture Notes in Computer Science*, pages 499–513. Springer, 2009.
- [MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.
- [MH06] Shahar Maoz and David Harel. From Multi-Modal Scenarios to Code: Compiling LSCs into AspectJ. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2005, Portland, Oregon, USA, November 5-11, 2006*, pages 219–230, 2006.
- [MHK02] Rami Marelly, David Harel, and Hillel Kugler. Multiple Instances and Symbolic Variables in Executable Sequence Charts. In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*, volume 37 of *ACM SIGPLAN Notices*, pages 83–100, November 2002.
- [MOF06] Meta Object Facility (MOF) Core Specification, January 2006. OMG document `formal/06-01-01`.
- [MWW08] Björn Metzler, Heike Wehrheim, and Daniel Wonisch. Decomposition for compositional verification. In *ICFEM '08: Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, pages 105–125, Berlin, Heidelberg, 2008. Springer-Verlag.
- [MZ03] Thomas Maier and Albert Zündorf. The Fujaba Statechart Synthesis Approach. In *Proceedings of the 2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, ICSE 2003*, 2003.
- [OCL10] Object Constraint Language (OCL 2.2), February 2010. OMG document `formal/2010-02-01`.
- [PGZ05] Cory Plock, Benjamin Goldberg, and Lenore Zuck. From Requirements to Specifications. In *Proceedings of the 12th Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*. IEEE Computer Society Press, 2005.
- [Pnu85] Amir Pnueli. In Transition from Global to Modular Temporal Reasoning about Programs. *Logics and models of concurrent systems*, pages 123–144, 1985.

- [Poh07] Klaus Pohl. *Requirements Engineering. Grundlagen, Prinzipien, Techniken*. Dpunkt.Verlag GmbH, 2007.
- [QVT08] Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, v1.0, April 2008. OMG document `formal/08-04-03`.
- [RJB05] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Professional, 2 edition, May 2005.
- [Sch94] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In Tinhofer, G. (Ed.), editor, *20th Int. Workshop on Graph-Theoretic Concepts in Computer Science, Lecture Notes in Computer Science (LNCS)*, volume Volume 903, pages 151–163, Heidelberg, 1994. Springer Verlag.
- [Sta85] Eugene W. Stark. A proof technique for rely/guarantee properties. *Foundations of Software Technology and Theoretical Computer Science*, 206:369–391, 1985.
- [SUB08] German Sibay, Sebastian Uchitel, and Victor Braberman. Existential Live Sequence Charts Revisited. In *Proceedings of the ACM/IEEE 30th Int. Conference on Software Engineering ICSE '08*, pages 41–50, 2008.
- [Sys08] OMG Systems Modeling Language (OMG SysML 1.1), December 2008. OMG document `ormsc/08-11-02`.
- [UBC07] Sebastian Uchitel, Greg Brunet, and Marsha Chechik. Behaviour Model Synthesis from Properties and Scenarios. In *Proceedings of the 29th international Conference on Software Engineering, ICSE '07*, pages 34–43, Washington, DC, USA, 2007. IEEE Computer Society.
- [UBC09] Sebastian Uchitel, Greg Brunet, and Marsha Chechik. Synthesis of Partial Behavior Models from Properties and Scenarios. *IEEE Transactions on Software Engineering*, 35(3):384–406, 2009.
- [UK01] S. Uchitel and J. Kramer. A Workbench for Synthesising Behaviour Models from Scenarios. In *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001*, pages 188–197, 2001.
- [UKM03] S. Uchitel, J. Kramer, and J. Magee. Synthesis of Behavioral Models from Scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, 2003.
- [UML09] UML 2.2 Superstructure Specification, February 2009. OMG document `formal/2009-02-02`.
- [VDI04] VDI, Verein Deutscher Ingenieure. *VDI Guideline 2206, Design Methodology for Mechatronic Systems*. Berlin, 2004.
- [vL09] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

- [Wag06] Robert Wagner. Developing Model Transformations with Fujaba. In Holger Giese and Bernhard Westfechtel, editors, *Proceedings of the 4th International Fujaba Days*, volume tr-ri-06-275 of *Technical Report*, pages 79–82, Bayreuth, Germany, September 2006. University of Paderborn.
- [Wag09] Robert Wagner. *Inkrementelle Modellsynchronisation*. PhD thesis, University of Paderborn, February 2009. (in German).
- [WS00] Jon Whittle and Johann Schumann. Generating statechart designs from scenarios. In *Proceedings of the 22nd international conference on Software engineering, ICSE '00*, pages 314–323, 2000.
- [WS02] Jon Whittle and Johann Schumann. Statechart Synthesis from Scenarios: an Air Traffic Control Case Study. In *In Proceedings of the International Workshop on Scenarios and State Machines, ICSE '02*, 2002.

List of Figures

1.1	A V-model describing the development of a mechatronic system, on the basis of the VDI 2206 “Design Methodology for Mechatronic Systems”	3
1.2	The RailCab system: On-demand transportation of passengers and goods by autonomous rail vehicles (taken from [ADG ⁺ 09])	4
1.3	Three use cases of RailCabs entering a merging switch	5
2.1	Different interactions on the NMS level of the RailCabs system.	12
2.2	Overview of the diagram kinds used during the conceptual design of mechatronic systems (taken from [ADG ⁺ 09])	14
2.3	The informal description, active structure diagram and activity diagram for the use case Drive onto track section	16
2.4	The informal description, active structure diagram and activity diagram for the use case Drive onto merging switch	17
2.5	An illustration of the time constraints formulated in the “Drive onto merging switch” use case.	19
3.1	The object system and MSDs with concrete lifelines	24
3.2	An extended version of the MSD RequestEnterAtEndOfTrackSection	28
3.3	The Büchi automation corresponding to the iterative interpretation of the universal MSD RequestEnterAtEndOfTrackSection in Fig. 3.2.	28
3.4	An illustration of valid and violating super-steps.	31
3.5	A message parameter specified by an expression over an object property	35
3.6	An example for a message with a side-effect on the receiving object	35
3.7	Simple examples of assignments and conditions in MSDs.	37
3.8	The MSD ReplyOfSwitchControlNotTooEarly.	38
3.9	An example the RequestEnterAtEndOfTrackSection with symbolic lifelines in a dynamic system: which object does a lifeline represent?	40
3.10	A class model of the RailCab system and a possible object system (an object diagram representation of the system shown in Fig. 3.9)	43
3.11	The MSD RequestEnterAtEndOfTrackSection with symbolic lifelines and a binding expression	44
3.12	The MSD RequestEnterAtEndOfTrackSection with additional forbidden messages	46

3.13	An example of a light switch modeled as a system of Timed Automata in UPPAAL	48
3.14	An example of an axiom.	52
3.15	A TGG rule as a graph grammar production rule.	53
3.16	The short-hand notation of a TGG-rule with additional attribute value constraints.	54
3.17	The creation of an instance of the axiom in the model.	55
3.18	Interpretation of a TGG rule in a forward transformation scenario.	55
3.19	Result of the forward application of the TGG rule <code>CollaborationToSystemATAndEnvironmentAT</code>	56
4.1	The principle of encoding an MSD specification as a system of Timed Game Automata	59
4.2	An untimed MSD specification, formalizing the requirements of the use case <code>Drive onto track section</code>	61
4.3	The environment and system automata for an untimed example MSD specification	64
4.4	The TGA for the MSD <code>RequestEnterAtEndOfTrackSection</code>	69
4.5	The MSD <code>RequestEnterAtEndOfTrackSection</code> extended by an assignment and a condition.	74
4.6	The TGA for the MSD <code>RequestEnterAtEndOfTrackSection</code> extended by an assignment and a condition	74
4.7	The TGA for the assumption MSD <code>LastBreakBeforeEnterNext</code>	78
4.8	The specification of the use case <code>Drive onto merging switch</code> as an example to illustrate the MSD-to-TGA mapping for timed specifications	83
4.9	The environment and system automata for a timed example MSD specification	84
4.10	The automaton template for the MSD <code>ReplyOfSwitchControlNotTooEarly</code>	88
4.11	The automaton template for the assumption MSD <code>EnterNextAfterEightToTwelveSeconds</code>	93
4.12	A sketch of the compositional synthesis approach	100
4.13	A sketch of the production cell	104
4.14	Step 0 and Step 1 in the decomposition of the production cell specification	106
4.15	Splitting the lifeline of MSD <code>PressPlateAfterArmARelasesBlankPlate</code> according to Step 0	107
4.16	A sketch of how the compositional synthesis approach is mapped to the proof technique of Stark	111
5.1	The process of finding and resolving inconsistencies in scenario-based specifications envisioned in <code>SCENARIOTOOLS</code>	119
5.2	The collaboration diagram and MSDs of the <code>Warn RailCabs on track</code> use case specification	122
5.3	Illustration of how the play-out of the active MSDs belonging to a use case occurrence is aided by a synthesized controller strategy	126

5.4	The collaboration diagram and the (single) MSD specified for the use case Enter denied when hazard on next track section	129
5.5	A combination of the use cases Drive onto track section and Hazard occurred and Enter denied when hazard on next track section in a particular instance situation	129
5.6	The collaboration diagram and the (single) MSD for the use case Hazard occurred	130
5.7	The collaboration diagram for a use case that is composed of occurrences of the use cases Drive onto track section, Hazard occurred, and Enter denied when hazard on next track section	131
5.8	“Flattening” composed use cases to the input for the synthesis	132
5.9	An occurrence of the composed use case Enter denied when hazard on next track section AND Drive onto track section AND Hazard occurred that is executed based on a synthesized controller strategy	134
5.10	An example of incomplete and complete occurrences of a composed use case	136
5.11	An example of incomplete and complete occurrences of a composed use case under consideration of a context expression	138
6.1	The TGG rule MinimalEnvironmentMessage illustrates extensions to the TGG formalism.	144
6.2	The generalization hierarchy of the TGG rules for translating different kinds of messages.	146
6.3	An example where a specialized rule (according to Klar et al.) is applicable, but not the general rule.	148
6.4	An <i>attribute value constraint</i> is attached to a <i>slot node</i> and constrains the value of the <i>slot attribute</i> by the value specified by the <i>value expression</i>	150
6.5	An illustration of how many messages representing the same event are mapped to the same edge by different TGG rules	155
6.6	In the translation of composed use cases to a TGA network, the same MSD may have to be mapped to multiple MSD template automata if the use case occurs as an internal use case occurrence in the composed use case several times	158
7.1	A screenshot of the TGG rule editor	160
7.2	A transformation can be executed by a simple right-click action on an interpreter configuration file	161
7.3	The two-step transformation approach	161
7.4	The class models of the RailCab specification base package (simplified), and the use cases Drive onto track section and Drive onto merging switch	164
7.5	The merged package for use case Drive onto merging switch as shown in Fig. 7.4	165
7.6	The package merge relationships between the packages of some use case specifications in the RailCab system specification	166

7.7	An overview of the SCENARIOTOOLS modeling and simulation process	167
7.8	An overview of the MSD editor in SCENARIOTOOLS and the simulation user interface.	168
7.9	A screenshot of the user interface of the 3D visualization that was realized within the SCENARIOTOOLS VISUALIZATION project	170
A.1	An ECore meta-model for UPPAAL TIGA (automaton templates, global/template declarations, system declarations)	192
A.2	An ECore meta-model for UPPAAL TIGA (functions, statements, expressions)	193
A.3	The UML-Profile for MSD-Specifications	194
A.4	The abstract syntax of an MSD specification: packages, classes, collaborations, parts, interactions, and lifelines	195
A.5	The abstract syntax of an MSD specification: messages, occurrence specifications, events, and operations	196
A.6	The abstract syntax of an MSD specification: conditions and expressions	197
B.1	An overview of the rule relationships in the TGG defining the MSD-to-TGA mapping	200
B.2	The TGG Axiom PackageToNta	204
B.3	Part of the TGG rule AbstractMSDSpecification (showing the environment automaton template created for an MSD specification as well as its instantiation)	205
B.4	Part of the TGG rule AbstractMSDSpecification (showing the system automaton template created for an MSD specification as well as its instantiation).	206
B.5	Part of the TGG rule AbstractMSDSpecification (showing the global variable and channel declarations that are created for an MSD specification)	207
B.6	Part of the TGG rule AbstractMSDSpecification (showing the global function declarations that are created for an MSD specification)	208
B.7	The TGG rule MSDSpecification (refines AbstractMSDSpecification)	209
B.8	The TGG rule TimedMSDSpecification (refines AbstractMSDSpecification)	210
B.9	Part of the TGG rule MSD (showing the MSD automaton template skeleton structure created for each MSD as well as the instantiation of the template)	211
B.10	Part of the TGG rule MSD (showing the template variable and function declarations created for an MSD automaton template)	212
B.11	Part of the TGG rule MSD (showing the global function declarations created for each MSD)	213

B.12	The TGG rule Lifeline (mapping a lifeline to the declaration of a lifeline variable and to structures which represent parts of MSD automaton template edge update and guard expressions; furthermore mapped to part of the <code>bool isHiddenEventEnabled(int player)</code> function body)	214
B.13	The TGG rule Message (mapping a message in a TGG to a globally declared constant, an edge in the MSD automaton template, and part of an expression that is declared for the MSD automaton template)	215
B.14	The TGG rule MinimalMessage (mapping a minimal message to different parts of edge of edge update and guard labels in the MSD automaton template as well as part of <code>bool enabled (int ev)</code> function that is declared for the MSD automaton template, refines the rule Message)	216
B.15	The TGG rule MinimalEnvironmentMessage (mapping a minimal environment message to an edge in the environment automaton template, refines the rule MinimalMessage)	217
B.16	The TGG rule MinimalSystemMessage (mapping a minimal system message to an edge in the system automaton template, refines the rule MinimalMessage)	217
B.17	The TGG rule ForbiddenMessage (mapping a forbidden message to parts of various function body expressions, refines the rule Message)	218
B.18	The TGG rule NonMinimalMessage (mapping a non-minimal message to an edge update label in the MSD automaton template as well as part of a body expression of the function <code>bool enabled (int ev)</code> in the MSD automaton template, refines the rule Message)	219
B.19	The TGG rule ColdMessage (mapping a cold message, refines the rule NonMinimalMessage)	220
B.20	The TGG rule ColdEnvironmentMessage (mapping a cold (monitored) environment message to the corresponding edge in the MSD automaton template, refines the rule ColdMessage)	220
B.21	The TGG rule ColdExecutedEnvironmentMessage (mapping a cold executed environment message to expression parts in global functions, refines the rule ColdEnvironmentMessage)	221
B.22	The TGG rule ColdSystemMessage (mapping a cold (monitored) system message to the corresponding edge in the MSD automaton template, refines the rule ColdMessage)	221
B.23	The TGG rule ColdExecutedSystemMessage (mapping a cold executed system message to expression parts in global functions, refines the rule ColdSystemMessage)	222
B.24	The TGG rule HotMessage (mapping a hot message, refines the rule NonMinimalMessage)	222
B.25	The TGG rule HotEnvironmentMessage (mapping a hot (monitored) environment message to the corresponding edge in the MSD automaton template, refines the rule HotMessage)	223

B.26	The TGG rule <code>HotExecutedEnvironmentMessage</code> (mapping a hot executed environment message to expression parts in global functions, refines the rule <code>HotEnvironmentMessage</code>)	223
B.27	The TGG rule <code>HotSystemMessage</code> (mapping a hot (monitored) system message to the corresponding edge in the MSD automaton template, refines the rule <code>HotMessage</code>)	224
B.28	The TGG rule <code>HotExecutedSystemMessage</code> (mapping a hot executed system message to expression parts in global functions, refines the rule <code>HotSystemMessage</code>)	224
B.29	The TGG rule <code>StateInvariant</code> (mapping a state invariant to an edge in the MSD automaton template and parts of global functions)	225
B.30	The TGG rule <code>Assignment</code> (mapping an assignment to a variable declaration in the corresponding MSD automaton template and adding an update expression to the according edge in the MSD automaton template, refines the rule <code>StateInvariant</code>)	225
B.31	The TGG rule <code>ClockReset</code> (mapping a clock reset to a clock declaration in the corresponding MSD automaton template and adding an update expression to the according edge in the MSD automaton template, refines the rule <code>StateInvariant</code>)	226
B.32	The TGG rule <code>Condition</code> (mapping a condition to an additional part of the corresponding edge's guard label as well as an additional edge that represents the violation of the condition, refines the rule <code>StateInvariant</code>)	226
B.33	TGG Rule <code>ColdCondition</code> (mapping a cold condition to an update label statement of the corresponding violating edge, refines the rule <code>Condition</code>)	227
B.34	The TGG rule <code>HotCondition</code> (mapping a hot condition to an update label statement of the corresponding violating edge, refines the rule <code>Condition</code>)	227
B.35	The TGG rule <code>ColdTimeCondition</code> (mapping a cold time condition to an update label statement of the corresponding violating edge, refines the rule <code>Condition</code>)	228
B.36	The TGG rule <code>HotMaximalDelay</code> (mapping a hot maximal delay to an update label statement of the corresponding violating edge, refines the rule <code>Condition</code>)	228
B.37	The TGG rule <code>HotMinimalDelay</code> (mapping a hot minimal delay to an edge representing the progression of the delay in the corresponding MSD automaton template as well as parts in global functions)	229
C.1	A class diagram illustrating the merge relationships between the different use case specification packages	233
C.2	The class model defined in the package <code>RailCabBase</code>	233
C.3	The collaboration diagram in the <code>Drive onto track</code> section use case specification	235
C.4	The MSD <code>RequestEnterAtEndOfTrackSection</code> of the use case specification <code>Drive onto track</code> section	236

C.5	The MSD <code>SetDefaultNextTSC</code> of the use case specification Drive onto track section	237
C.6	The MSD <code>DefaultEnterAllowed</code> of the use case specification Drive onto track section	237
C.7	The MSD <code>StopWhenEnterDenied</code> of the use case specification Drive onto track section	237
C.8	The class diagram in the Drive onto track section use case specification	238
C.9	The collaboration diagram in the Drive onto branching switch use case specification	239
C.10	The class diagram in the Drive onto branching switch use case specification	239
C.11	The MSD <code>EnterDeniedWhenHazardOnBranchingTrackSection</code> of the use case specification Drive onto branching switch	240
C.12	The MSD <code>NotifyRailCabWhenHazardOnSubsequentBranching-SwitchResolved</code> of the use case specification Drive onto branching switch	241
C.13	The MSD <code>SetNextTrackSectionControlWhenBranchingOnSwitch</code> of the use case specification Drive onto branching switch	242
C.14	The RailCab's energy management requires the communication between different modules in the RailCab (AMS level) (taken from [ADG ⁺ 09])	243
C.15	The collaboration diagram in the RailCab Energy Management use case specification	244
C.16	The class diagram in the RailCab Energy Management use case specification	244
C.17	The MSD <code>SetNextTrackSectionControlWhenBranchingOnSwitch</code> of the use case specification Drive onto branching switch	246
C.18	The ECore class model resulting from the UML-to-Ecore transformation of the package <code>RailCabIntegrated</code>	247
C.19	An instance model of a simple track system with five track sections, two switches and two RailCabs	248
C.20	A screenshot of the SCENARIOTOOLS use interface	249
C.21	A sketch of the situation described by the RailCab Obstacle Detected use case	250
C.22	The collaboration diagram of the Warn RailCabs on track use case specification	251
C.23	The MSD <code>WarningWhenObstacleDetected</code>	251
C.24	The MSD <code>ReportObstaclePosition</code>	251
C.25	The MSD <code>ReportObstaclePositionAndIssueWarning</code>	252
C.26	The MSDs that specify the requirements of the production cell.	261
C.27	The MSDs that specify the assumptions for the environment of the production cell.	262
C.28	Times of transformation and synthesis for the production cell specification with different concrete values for delays in the specification, in a setting where the system <i>is</i> allowed to delay active events.	263

C.29	Times of transformation and synthesis for the production cell specification with different concrete values for delays in the specification, in a setting where the system is <i>not</i> allowed to delay active events.	264
C.30	Part one of the decomposed specification – controller <u>c1</u> controls arm A.	265
C.31	Part two of the decomposed specification – controller <u>c2</u> controls the press and arm B.	267
C.32	Times of transformation and synthesis for the part specifications of the production cell with different concrete values for delays in the specification, in a setting where the system is allowed to delay active events.	268
C.33	Times of transformation and synthesis for the part specifications of the production cell with different concrete values for delays in the specification, in a setting where the system is <i>not</i> allowed to delay active events.	269
C.34	Example of an untimed MSD specification with exponentially many activations of MSDs (here $n = 3$)	272
C.35	A table listing the time for transforming and synthesizing strategies for an untimed MSD specification with exponentially many MSD activations.	272
C.36	A chart visualizing the time for transforming and synthesizing strategies for an untimed MSD specification with exponentially many MSD activations.	273
C.37	Plotting the logarithm (\log_2) of the synthesis time against the size of the MSD specification	274
C.38	Example of a timed MSD specification with exponentially many activations of MSDs (here $n = 4$)	275
C.39	A table listing the time for transforming and synthesizing strategies for a timed MSD specification with exponentially many MSD activations.	275
C.40	A chart visualizing the time for transforming and synthesizing strategies for a timed MSD specification with exponentially many MSD activations.	276

Index

- active structure diagram, 13
- admissible implementation, *see* MSD specification, admissible implementation
- assignment, *see* Modal Sequence Diagrams (MSDs), assignment
- assume-guarantee paradigm, 8, 98
 - soundness, 98
- automaton template, *see* UPPAAL, (automaton) template
- autonomous mechatronic system (AMS), 12

- bound (use case occurrence), *see* composed use case, bound use case occurrence

- clock (Timed Automata), *see* Timed Automata, clock variables
- clock (timed MSDs), *see* Modal Sequence Diagrams (MSDs), clock
- cold
 - cut, *see* cut, temperature
- cold violation, 26
- complete (composed use case occurrence), *see* composed use case, complete composed use case occurrence
- complete controller (complete winning strategy), 126
- composed use case specification, 118
- composed use cases, 186
 - bound use case occurrence, 135
 - complete composed use case occurrence, 135
 - levels of composed use case occurrences, 139
 - partially bound composed use case occurrence, 135
 - unified use case occurrences, 139
- compositional synthesis technique, 97, 186
 - example (production cell), 263
 - global specification, 97
 - part specification, 99
 - soundness, 108
- compositional verification, 97

- conceptual design, 2
- condition, *see* Modal Sequence Diagrams (MSDs), condition
- consistent, *see* MSD specification, consistent
- consistent executability, *see* MSD specification, consistently executable
- context expressions, 137
- cut, 26
 - execution kind, 26
 - temperature, 26

- disciplines, 11

- edges (UPPAAL), *see* UPPAAL, edges
- environment, *see* MSD specification, environment objects
- environment automaton template, 62
- event, 25
 - environment event, 25
 - synchronous, 25
 - system event, 25
- executed
 - cut, *see* cut, execution kind

- first event, *see* Modal Sequence Diagrams (MSDs), first event

- global declarations (UPPAAL), *see* UPPAAL, global declarations
- global specification, *see* compositional synthesis technique, global specification

- hidden event, 37
- hot
 - cut, *see* cut, temperature
- hot violation, 26

- implementation (development phase), 3
- incomplete (composed use case occurrence), *see* composed use case, partially bound composed use case occurrence

- levels (of composed use case occurrences),
 see composed use cases, levels of
 composed use case occurrences
- lifeline, 19
 - binding expressions, 42
 - locations, 26
 - symbolic, 39
- lifeline variables, 67
- Live Sequence Charts (LSCs), 6, 20
- local declarations (UPPAAL), *see* UPPAAL,
 local declarations
- locations (UPPAAL), *see* UPPAAL, locations
- maximal delay, *see also* Modal Sequence
 Diagrams (MSDs), time condi-
 tions
- mechatronic function modules (MFM), 11
- Mechatronic Systems, 11
- message, 23, 25
 - enabled, 26
 - forbidden, 45
 - side-effect, 35
 - temperature, 26
 - unifiable, 25
- Message Sequence Charts (MSCs), 19
- messages
 - symbolic, 171
- minimal delay, *see also* Modal Sequence Di-
 agrams (MSDs), time conditions
- minimal event, *see* Modal Sequence Di-
 agrams (MSDs), first event
- Modal Sequence Diagrams (MSDs), 6, 21,
 23
 - active, 26
 - active copy, *see* active
 - anti-scenario, 38
 - assignment, 35
 - assumption MSDs, 8, 60, 186
 - clock (variable), 37
 - clock reset, 37
 - concrete, 39
 - condition, 35
 - diagram variable, 33
 - existential, 25
 - first event, 26
 - first message, 26
 - invariant interpretation, 27, 182
 - iterative interpretation, 27
 - symbolic, 39
 - time conditions, 37
 - maximal delay, 91
 - minimal delay, 91
 - UML profile, 193
- monitored
 - cut, *see* cut, execution kind
- MSD automaton template, 67
- MSD specification
 - admissible implementation, 30
 - consistent, 32, 79, 253
 - consistent (winning condition,
 changes to TGA model), 113
 - consistently executable, 32, 79, 253
 - environment objects, 23
 - system objects, 23
 - under-specified, 6
- MSD-to-TGA mapping, 57, 199
- networked mechatronic system (NMS), 12
- Object Constraint Language (OCL), 9, 149
- object system, 23
- part specification, *see* compositional syn-
 thesis technique, part specifica-
 tion
- partial observability, 115
- partially bound (composed use case occur-
 rence), *see* composed use case,
 partially bound composed use
 case occurrence
- play-out algorithm, 6, 21, 177
- principle solution, 3
- production cell, 257
- production cell example, 103
- progressing edge (MSD automaton tem-
 plate), 75
- RailCab, 4
- role binding, *see* use case occurrence, role
 binding
- run, 25
- safety violation, 26
- scenarios, 3
- self-optimizing systems, 1
- smart play-out, 177
- super-step, 177
- synchrony hypothesis, 30, 38
- system, *see* MSD specification, system ob-
 jects
- system automaton template, 62
- system definition (UPPAAL), *see* UPPAAL,
 system definition
- Systems Modeling Language (SysML), 13
- template (Timed Automaton), *see* UP-
 PAAL, (automaton) template
- Timed Automata, 45
 - clock variables, 46
- Triple Graph Grammars (TGGs), 9, 50,
 143
 - abstract rule, 145
 - application conditions, 151
 - application scenarios, 51
 - attribute constraints, 149
 - attribute value constraints, 54

- context edge, 53
 - context node, 53
 - correspondence graph, 51
 - domain model, 51
 - edge binding, 52
 - forward transformation, 51
 - generalization, 144
 - global constraints, 155
 - inheritance, *see* generalization
 - match (pattern), 52
 - node binding, 52
 - produced edge, 53
 - produced node, 53
 - refined nodes, 143
 - reusable nodes, 56, 144, 152
 - reusable pattern, 156
 - rule generalization, 55, 187
 - stereotype constraints, 152
- UPPAAL TIGA, 46, 48
- (un)controllable edge, 48
 - ECore meta-model, 191
 - winning states (synthesis algorithm), 49
- UPPAAL, 46
- (automaton) template, 46
 - channels, 47
 - edges, 47
 - enabled, 47
 - guard label, 47
 - synchronization label, 47
 - update label, 47
 - global declarations, 46
 - local declarations, 47
 - locations, 47
 - committed, 47
 - urgent, 47
 - system definition, 46
- unified (use case occurrences), *see* composed use case, unified use case occurrences
- use case occurrence, 123
- role binding, 123
- use cases, 3
- composed, 9
- violating edge (MSD automaton template), 75
- visible event, 37
- winning states, *see* UPPAAL TIGA, winning states